VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

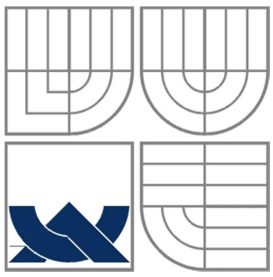
LADICÍ NÁSTROJ PRO VÍCEPROCESOROVÝ SYSTÉM NA ČIPU

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

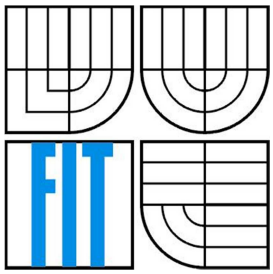
AUTOR PRÁCE
AUTHOR

Bc. MICHAL ŠPAČEK

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

LADICÍ NÁSTROJ PRO VÍCEPROCESOROVÝ SYSTÉM NA ČIPU

DEBUGGER FOR MULTIPROCESSOR SYSTEM ON A CHIP

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MICHAL ŠPAČEK

VEDOUCÍ PRÁCE
SUPERVISOR

prof. Ing. Tomáš Hruška, CSc.

BRNO 2011

Abstrakt

Projekt Lissom se zabývá metodologií hardware/software co-design. V rámci tohoto projektu bylo vyvinuto prostředí pro návrh víceprocesorových systémů na čipu. Prostředí umožnuje i návrh aplikací pro víceprocesorové systémy. Součástí prostředí je i ladicí nástroj, který umožňuje ladění aplikací pro jednoprocesorové systémy. V této práci je vývojové prostředí popsáno a je navrženo a implementováno rozšíření stávajícího ladicího nástroje o možnosti ladění víceprocesorových systémů na základě požadavků standardu Nexus.

Abstract

The Lissom project deals with the hardware-software co-design methodology. In this project, an integrated desktop environment for a design of multiprocessor systems on chip was developed. This environment can be used also for developing applications for multiprocessor systems. One part of the environment is a debugger that can be used to debug single core systems. In this thesis, a single processor debugger tool is described in detail and an extension to this tool is proposed and implemented based on the Nexus standard. The extended debugger allows debugging of multiprocessor systems.

Klíčová slova

MPSoC, ladicí nástroj, Nexus

Keywords

MPSoC, debugger, Nexus.

Citace

Špaček Michal: Ladicí nástroj pro víceprocesorový systém na čipu, semestrální projekt, Brno, FIT VUT v Brně, 2011

Ladicí nástroj pro víceprocesorový systém na čipu

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením prof. Ing. Tomáše Hrušky, CSc.
Další informace mi poskytl Ing. Zdeněk Přikryl.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Michal Špaček
25. 05. 2011

Poděkování

Především bych rád poděkoval mému konzultantovi Ing. Zdeňku Přikrylovi za poskytnutou pomoc, odborné vedení a čas věnovaný při konzultacích k tématu mé diplomové práce

Obsah

1 Úvod

V posledních letech jsme byli svědky ohromného nárůstu poptávky a výroby vestavěných systémů, které se uplatňují v nejrůznějších oblastech. Jádrem takového systému může být systém na čipu (SoC - System-on-chip), což je elektronický systém, který je implementován na jediném integrovaném obvodě (čipu) a přesto obsahuje veškeré nutné části pro svůj chod. Mezi tyto části patří mikroprocesor, bloky pamětí, časovač, systémový řadič a řadiče pro externí rozhraní jakou jsou standardní USB, Ethernet apod. Všechny tyto části jsou obvykle uvnitř čipu propojeny interní sběrnicí. Pokud systém na čipu obsahuje více procesorů, mluvíme o víceprocesorovém systému na čipu (MPSoC - multiprocessor System-on-Chip). Umístění všech těchto částí do jednoho čipu má i cenový význam – vyrobit jeden čip je mnohem jednodušší a levnější a ve výsledku má i daný produkt menší velikost, jeden čip zabírá méně prostoru než deska s několika různými čipy.

Víceprocesorové systémy na čipu se nyní dostávají do popředí zejména z důvodu použití *procesorů s aplikačně specifikou instrukční sadou* (ASIP – *application specific instruction set processor*), jejichž instrukční sada je vysoce optimalizovaná pro efektivní plnění konkrétní úlohy (např. pro akceleraci vykreslování 3D grafiky, zpracování zvuku apod.). Tato vysoká optimalizace snižuje spotřebu čipu i potřebnou plochu čipu, který daný procesor na čipu zabírá, a proto může být na čipu umístěno více těchto procesorů. Procesory mohou být optimalizovány na jiné úlohy a výsledný systém je pak několikrát výkonnější, než kdyby se použil systém na čipu s jedním výkonným, ale obecným mikroprocesorem.

Samotný čip ovšem je jen jedna součást systému na čipu. Druhou nezbytnou součást tvoří programové vybavení (software) – tedy programy, které běží na jednotlivých procesorech. Protože instrukční sada těchto procesorů je vysoce optimalizovaná, je vhodné navrhovat software pro tyto procesory souběžně s návrhem instrukční sady. Touto problematikou se zabývá metodologie Hardware-software co-design – jde tedy o souběžný návrh technického i programového vybavení.

Projekt Lissom je projekt, jehož cílem je vytvořit jednotné prostředí pro návrh takovýchto systémů na čipu, včetně jejich programů. Pro návrh a modelování architektury systému na čipu používá jazyk ISAC. Podle modelu, který je popsán pomocí jazyka ISAC [1], lze pak automaticky generovat další nástroje pro tvorbu programů pro konkrétní systémy.

Protože při vývoji aplikací vždy vznikají chyby, je vhodné mít nástroj, který pomáhá tyto chyby lokalizovat a odstranit. Tento nástroj se nazývá ladicí nástroj neboli debugger. V rámci této práce pak byl rozšířen nástroj pro ladění aplikací pro víceprocesorové systémy na čipu prostřednictvím generického simulátoru.

1.1 Struktura práce

Jako první cíl této práce lze považovat seznámení se s architekturou víceprocesorových systémů na čipu, které jsou popsány v první kapitole. Dále následuje seznámení se se simulační platformou projektu Lissom a principy a nástroji v něm používanými. Toto je popsáno v druhé kapitole. Ve čtvrté kapitole je detailněji popsán ladicí nástroj projektu Lissom, který se používá pro ladění jednoprocesorových systémů. Ve páté kapitole je představen standard Nexus 5001 [2], který popisuje ladicí rozhraní pro víceprocesorové systémy na čipu. Funkcionalita ladicího nástroje projektu Lissom je zde porovnána s požadovanou funkcionalitou standardu Nexus 5001. V šesté kapitole je popsán hlavní cíl této práce a to návrh a implementace rozšíření pro ladicí nástroj projektu Lissom, který přidává funkcionalitu pro ladění víceprocesorového systému na čipu. V sedmé kapitole je pak nový

ladicí nástroj testován a výsledky ladění a simulace jsou prezentovány ve formě grafů. V osmé kapitole je navíc prezentován návrh rozšíření ladicího nástroje tak, aby byl kompatibilní s rozhraním Nexus. Ladicí nástroj tak bude umožňovat bezproblémové ladění programů jak ve víceprocesorových simulacích, tak i v reálných implementacích v hardware.

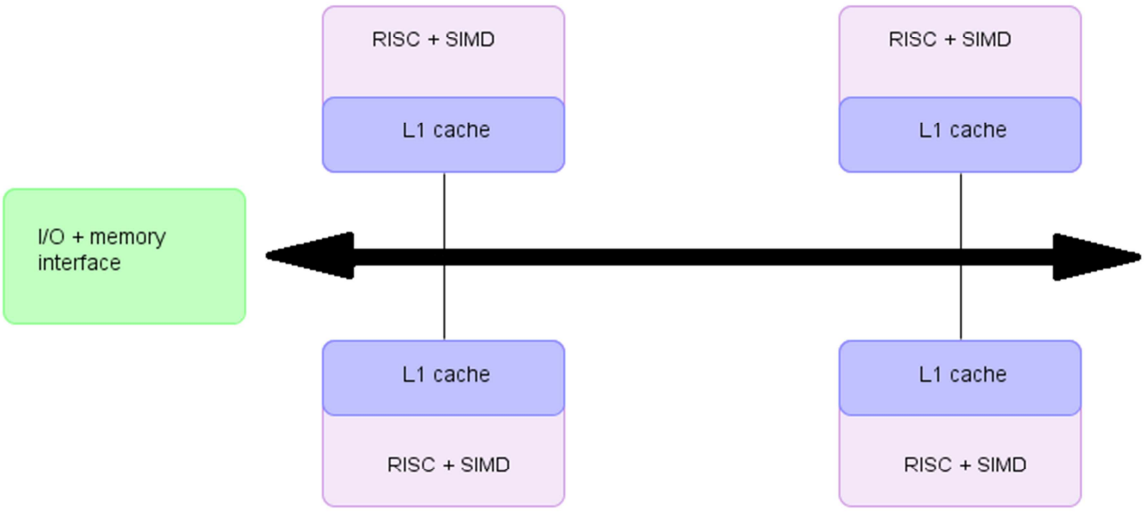
2 Architektura víceprocesorových systémů na čipu

2.1 Architektury se sdílenou sběrnicí

Architektura víceprocesorových systémů na čipu je značně závislá na použití daného systému. První víceprocesorový systém byl navržen pro identické zpracovávání signálů několika datových kanálů [3]. Protože zpracování každého kanálu je identické, tak i architektura tohoto víceprocesorového systému na čipu je homogenní. To znamená, že obsahuje pouze stejné procesory. Příklad takové architektury je znázorněn na obrázku 2.1.

Mnohem větší skupinu víceprocesorových systémů na čipu tvoří heterogenní systémy, ve kterých jsou použity na čipu různé procesorové jádra, které jsou obvykle vysoce specializované pro danou úlohu. Příkladem použití takového systému jsou například systémy určené pro zpracování paketů v počítačových sítích. Dalším možným příkladem použití jsou systémy pro zpracování multimédií, které se kromě různých jednoúčelových přehrávacích zařízení rozšiřují i do zařízení jako jsou mobilní telefony nebo tablety. Zástupcem takového systému na čipu je například víceprocesorový systém na čipu Nvidia Tegra 2, který obsahuje dvě jádra procesoru ARM Cortex A9, procesor pro dekódování videa ve vysokém rozlišení, procesor pro enkódování videa ve vysokém rozlišení, procesor pro zpracování zvuku, procesor pro zpracování obrazu a také i dedikovaný grafický procesor pro zpracování 2D a 3D grafiky.

Návrh propojení jednotlivých procesorových jader systému na čipu vycházel z konvenčních víceprocesorových systémů a jejich snahy integrovat je do jediného čipu. Proto se i ve většině víceprocesorových systémů používají sdílené sběrnice nebo rodiny sběrnic, které jsou propojeny mosty. Nejznámějšími takovými sběrnicemi je ARM AMBA [4] a IBM CoreConnect [5].



Obrázek 2.1: Příklad architektury se sdílenou sběrnicí

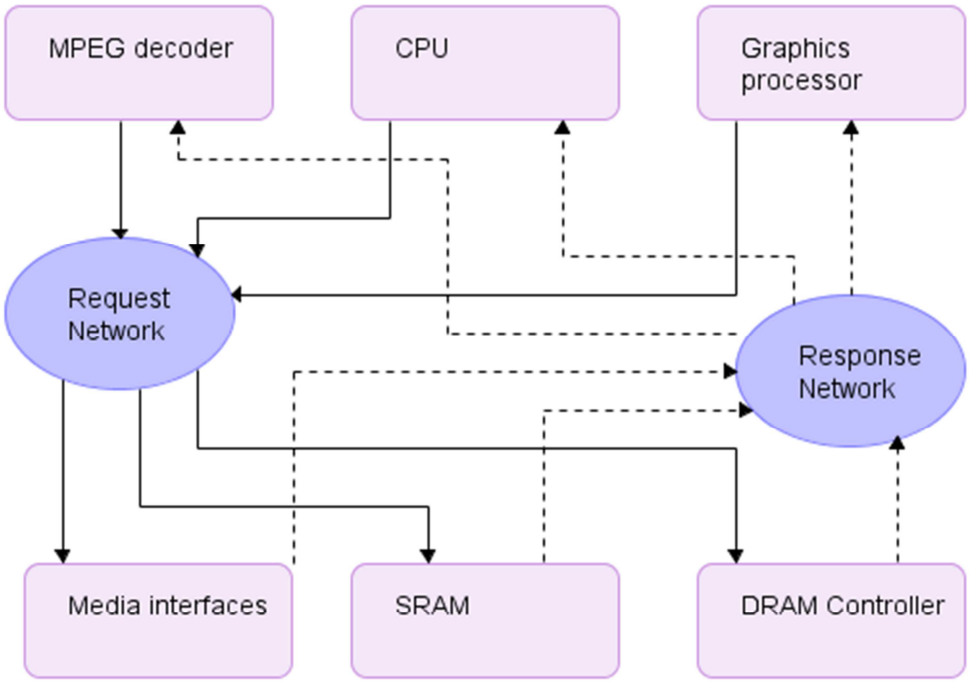
Příkladem takové architektury může být Vícejádrový systém na čipu pro dekódování streamovaného MPEG-4 videa[6], který obsahuje VLIW procesor nazvaný Macro block engine, RISC procesor s upravenou instrukční pro podporu zpracování proudového videa a volitelně zvukový DSP procesor. Použití těchto procesorových jader bylo zvoleno podle specifických požadavků algoritmu dekódování videa ve formátu MPEG-4.

2.2 Architektura NoC

Předpokládá se, že růst komplexnosti víceprocesorových systému na čipu bude růst s narůstajícím počtem procesorů a zpracovávajících bloků. Současné architektury založené na sběrnicích se pak budou potýkat s nedostatkem výkonu a zároveň spotřebovávat více energie aby byla dosažena dostatečná úroveň požadované rychlosti komunikace uvnitř čipu. Z toho důvodu se začala hledat alternativní architektura. Jednou z takových alternativních architektur je síť na čipu (*network-on-chip* - NoC)[7].

Základní ideou této architektury je použití hierarchických sítí se směrovači, které umožní mnohem efektivnější tok paketů od zdrojů k cílům. Zároveň tak bude umožněno i současné použití více komunikačních kanálů. Tato architektura oproti architektuře se sdílenou pamětí je energeticky úspornější a zároveň i výkonnější.

Výkonost této architektury je založena na síťové topologii, kdy jednotlivé spoje můžou být současně využity pro různé pakety. Ovšem tohoto paralelismu bude dosaženo jen tehdy, pokud bude i daný algoritmus navržen tak, aby využil tento potenciál.



Obrázek 2.2: Příklad architektury NoC

3 Simulační platforma Lissom

3.1 Jazyky pro popis architektur

Jazyky pro popis architektur (Architecture Descrption Language - ADL) [8] vznikly pro podporu automatizace návrhu vestavných procesorů. Z modelu vytvořeného pomocí specifikace v ADL lze automaticky generovat nástroje jednak pro tvorbu prototypů hardwaru a také i softwarové nástroje jako překladač a simulátor. Tyto jazyky lze rozdělit do tří kategorií podle druhu obsahu, na jehož popis se zaměřují.

První skupinou jsou strukturální jazyky pro popis architektur. Tyto jazyky umožňují přesně popsat strukturu pomocí architektonických komponent a jejich propojení, ale je u těchto jazyků problém extrahovat popis instrukční sady. Mezi tyto jazyky patří například jazyk MIMOLA [9]

Druhou skupinou jsou jazyky behaviorální, které se snaží abstrahovat informace o chování od detailní struktury. Behaviorální jazyky explicitně popisují sémantiku instrukcí, přičemž ignorují detailní hardwarové struktury. Mezi tyto jazyky patří například jazyky nML [10] a ISDL [11].

Poslední skupinou jsou smíšené jazyky pro popis architektur, které se snaží popsat jak strukturální tak i behaviorální detaily dané architektury. Mezi tyto jazyky patří například jazyk LISA (Language for Instruction Set Architecture) [12], ze kterého vychází i jazyk ISAC [1] vyvinutý v rámci projektu Lissom

3.1.1 Jazyk ISAC

Jazyk ISAC je jazyk pro popis architektur MPSoC (Multiprocessor Systems on a Chip), přesněji pro popis architektur procesorů s aplikačně specifickou instrukční sadou. Jazyk ISAC vychází z jazyka LISA a patří mezi smíšené jazyky pro popis architektur.

Model v jazyce ISAC slouží jako vstup pro generátory, jejichž úkolem je automatizovat vytváření prostředí pro vývoj a simulaci aplikací pro danou architekturu. Dalším cílem generátorů je automatizovat vytváření implementace dané architektury v hardwaru, to je vytvořit implementaci v jazyku pro popis hardware (např. VHDL). Generátory lze tedy rozdělit do dvou skupin: generátory softwarových nástrojů (assembler, linker, simulátor…) a generátory implementující vytváření hardwarové implementace architektury. Jako vstup těchto generátorů ovšem neslouží přímo zdrojové soubory v jazyce ISAC, ale tyto soubory jsou nejprve předzpracovány a přeloženy specializovaným překladačem do formy metadat ve formátu XML. Výstupem generátorů jsou pak různé zdrojové soubory – pro softwarové nástroje jsou to zejména soubory se zdrojovým kódem v C++, pro hardwarovou implementaci zdrojové soubory ve VHDL.

Samotný model architektury procesoru v jazyce ISAC lze rozdělit do několika částí podle druhu popisovaných informací o architektuře.

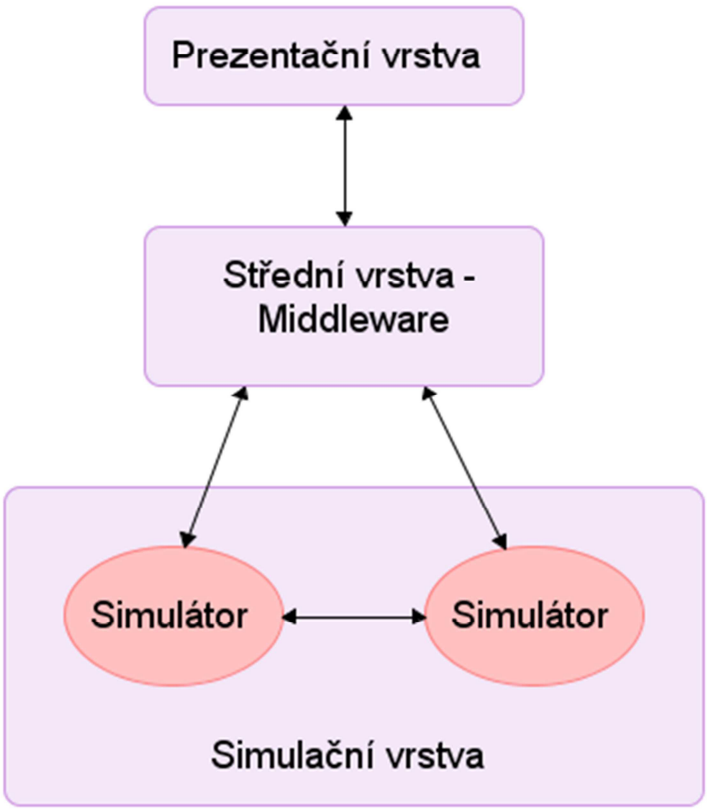
První z těchto částí je model instrukční sady. Ten je popsán množinou takzvaných operací, které popisují chování jednotlivých atomických operací v rámci architektury procesoru. U jednotlivých instrukcí je popsán jejich popis pro jazyk symbolických instrukcí a způsob jejich kódování do binární reprezentace. Také je zde popsáno chování instrukcí, k tomuto účelu je možné použít podmnožinu jazyka C. Instrukční model tedy popisuje dva jazyky – jazyk symbolických instrukcí a strojového kódu. Pomocí modelu instrukční sady pak lze vytvořit nástroje, které dokáží převádět programy z jednoho jazyka do druhého – nástroje assembler a disassembler.

Další částí modelu architektury procesoru je část, kde jsou popsány zdroje procesoru. Zdroji se rozumí registry, sběrnice, mapa paměti a cache. U všech zdrojů lze nastavit množství parametrů jako je velikost, velikost bloku, pořadí bytů ve slově paměti, latence apod.

Další důležitá část modelu je časový model. Tento model popisuje množinu událostí (množinu spuštěných modelů chování), které jsou spuštěny pro každý hodinový cyklus. Tento model popisující detailní časování umožňuje vytvořit simulátor založený na jednotlivých cyklech, jehož funkcionalita se nejvíce blíží reálnému hardwaru. Na základě tohoto modelu se dá poté vygenerovat implementace řadiče mikroprocesoru.

3.2 Třívrstvá architektura

Cílem projektu Lissom je vytvořit komplexní prostředí pro návrh a vývoj procesorů, které bude zároveň sloužit i pro návrh a vývoj aplikací pro tyto procesory. Vzhledem k velké komplexnosti je architektura projektu Lissom rozdělena na tři vrstvy: prezentační, střední vrstva (*middleware*) a simulační vrstvu. Tyto vrstvy jsou nezávislé (programy reprezentující tyto vrstvy mohou běžet na různých počítačích) a komunikují mezi sebou pomocí předem definovaného protokolu. Následující obrázek znázorňuje jednotlivé vrstvy a komunikaci mezi nimi.



Obrázek 3.1: Třívrstvá architektura

3.2.1 Prezentační vrstva

Prezentační vrstva zajišťuje interakci uživatele s celým systémem. Připojuje se ke střední middleware vrstvě, které přeposílá příkazy a zdrojové soubory ke zpracování. Výsledek zpracování pak zobrazuje uživateli.

Prezentační vrstva je reprezentována dvěma základními nástroji. Jedním z nich je příkazová řádka, která přijímá a zobrazuje data pouze v textovém režimu. Její použití je vhodné, když je nutné dávkově zpracovat množství příkazů – například při automatickém testování procesorů pomocí sady předpřipravených programů. Příkazová řádka se při startu připojuje ke střední vrstvě, k čemuž používá nejčastěji protokol TCP/IP [13], ale je také možné použít mezi procesorovou komunikaci (IPC), ale tato technologie je dostupná pouze na systému Linux.

Druhým používaným nástrojem je grafické uživatelské rozhraní tvořené zásuvným modulem Eclissom pro platformu Eclipse. Toto prostředí nabízí mnohem pohodlnější práci pro uživatele, neboť kromě přeposílání příkazů střední vrstvě umožňuje také editaci zdrojových souborů (modelů procesoru nebo zdrojových kódů programů).

3.2.2 Střední vrstva – Middleware

Střední vrstva zajišťuje zpracování dat, které získá od prezentační vrstvy. Zejména se jedná o tyto činnosti: Překlad modelu procesoru z jazyka ISAC do popisu v XML, vygenerování nástrojů pro daný model procesoru (assembler, disassembler), vygenerování simulátoru daného procesoru, překlad zdrojových kódů programu pro daný procesor pomocí vygenerovaných nástrojů a nakonec i spuštění simulátorů.

Střední vrstva podporuje práci více uživatelů najednou a to tím způsobem, že pro každého uživatele vytváří samostatný adresář, který obsahuje adresáře pro jednotlivé projekty, do kterých se ukládají generované nástroje a dočasné soubory.

3.2.3 Simulační vrstva

Simulační vrstva je tvořena jednotlivými simulátory, které vygeneruje střední vrstva podle konkrétního modelu. Samotné simulátory jsou pak reprezentovány přeloženými programy, které můžou běžet na stejném počítači jako střední vrstva. V tomto případě střední vrstva jenom tyto programy spustí. Ale existuje i možnost spuštění simulátorů na různých počítačích a potom je nutné, aby střední vrstva simulátory na tyto počítače nejprve nainstalovala. K tomuto účelu střední vrstva používá SCP[14]. Po nainstalování střední vrstva předá každému simulátoru seznam všech simulátorů, aby všechny simulátory byly schopny mezi sebou navázat spojení. Toto je nutné zejména u simulace víceprocesorového systému, u simulace jednoprocesorového systému běží jen jeden simulátor a ten komunikuje pouze se střední vrstvou. Při simulaci víceprocesorového systému je pro každý procesor vytvořen samostatný simulátor. Tyto simulátory pak mohou být spuštěny na různých počítačích, čímž může být simulace urychlena.

Samotná víceprocesorová simulace může běžet synchronně nebo asynchronně. Při synchronní simulaci musí být jednotlivé simulátory mezi sebou synchronizované. Při této variantě první simulátor generuje hodinový signál a ostatní simulátory vždy čekají na tento signál a až poté provedou jeden krok simulace. Při asynchronní simulaci se tato synchronizace nepoužívá a všechny simulátory běží nezávisle, jen střední vrstva čeká na dokončení práce všech simulátorů.

Samotná komunikace mezi simulátory pracuje na stejném principu jako komunikace se střední vrstvou a využívá speciální synchronizační protokol, pomocí kterého mohou jednotlivé simulátory přistupovat ke sdíleným zdrojům jiného procesoru.

Po startu každý simulátor načte simulovanou aplikaci ze spustitelného souboru a provede inicializaci procesoru. Samotná simulace začne až po obdržení zprávy od střední vrstvy.

Prezentační vrstva nemůže komunikovat přímo se simulátory, ale vždy k tomuto účelu využívá služeb střední vrstvy. Ta dané příkazy přeposílá konkrétním simulátorům. Střední vrstva také samozřejmě přeposílá zprávy od simulátoru zpět prezentační vrstvě.

Simulátor se zastaví, pokud dokončí simulaci daného programu nebo případně může běh simulace být pozastaven, pokud program narazí na breakpoint. Po ukončení všech simulátorů lze získat výsledky simulace, které zahrnují statistiky využití zdrojů procesoru během simulace, čas simulace a také frekvenci, která odpovídá času simulace a počtu cyklů procesoru.

3.3 Komunikační protokol

Jak již bylo zmíněno v poslední kapitole, vrstvy mezi sebou komunikují. K této komunikaci využívají komunikační protokol, který je založený na zjednodušeném značkovacím jazyku XML. Formát je zjednodušený v tom smyslu, že všechny značky jsou bez atributů a jsou vždy značeny velkými písmeny. Také nelze použít zkrácenou formu prázdné značky.

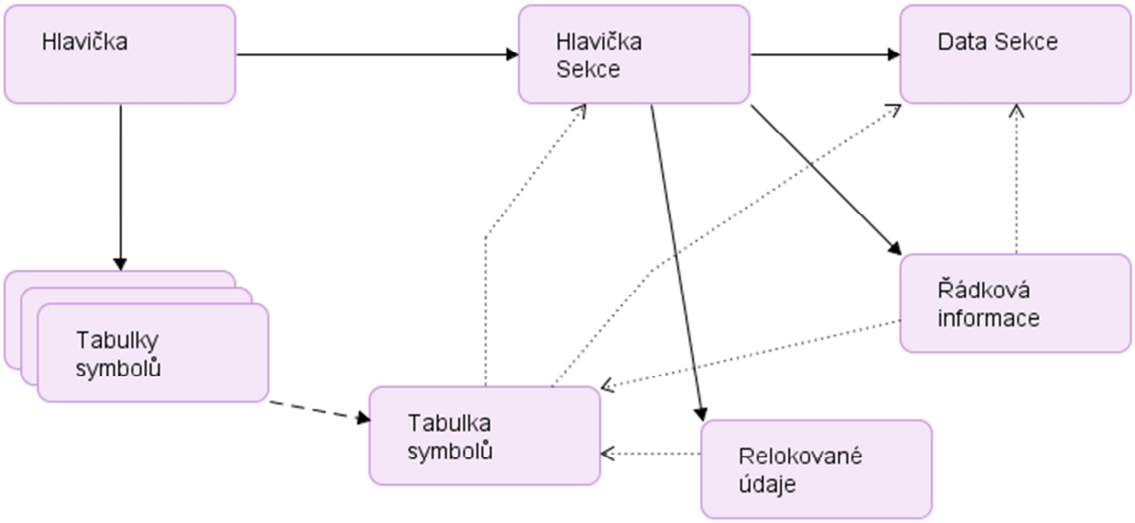
Samotný obsah zpráv přenášený pomocí komunikačního protokolu je tvořen příkazy s parametry. Příkaz je identifikován svým názvem. Za identifikací příkazu je připojen seznam argumentů. Jako argument může být zvolen i soubor, jehož zakódovaný obsah se přenáší jako samostatný parametr.

3.4 Formát spustitelného souboru projektu Lissom

Formát spustitelného soboru projektu Lissom [15] byl navržen s ohledem na maximální flexibilitu a nezávislost na konkrétní architektuře. Zejména šlo o tyto vlastnosti: nezávislost na šířce slova instrukce, nezávislost na způsobu uložení čísel, nezávislost na charakteru instrukční sady a nezávislost na způsobu uložení a druhu paměti a přístupu do ní.

Formát spustitelného souboru projektu Lissom je textový a rozděluje informace v něm uložené do několika sekcí. Všechny číselné hodnoty jsou uloženy vždy v desítkové soustavě, kromě číselných informací týkajících se kódu a inicializovaných dat, které jsou vždy uloženy ve dvojkové soustavě. Jedna hodnota je vždy uložena na jednom řádku. Jen textová hodnota je vždy uložena na dvou řádcích – první obsahuje délku textového řetězce a druhý obsahuje samotný řetězec.

Na začátku každého souboru je hlavička, která obsahuje magický řetězec, pomocí kterého lze rozeznat daný formát, dále základní údaje jako počet bitů a bytů ve slově, časové razítko, příznaky, počet sekcí, počet tabulek symbolů a absolutní adresa první tabulky symbolů. Každá sekce obsahuje další hlavičku obsahující informace o dané sekci, samotné binární data, údaje o číslech řádků obsahující informace návaznosti původních zdrojových souborů na relativní adresy a tabulku symbolů obsahující jména a typy symbolů, relativní adresy symbolů a případné doplňující informace.



Obrázek 3.2: Schéma zachycující strukturu objektového souboru

4 Ladicí program pro jednoprocesorový systém

Ladicí nástroj neboli debugger [16] je nástroj, který pomáhá určit, proč se daný program nechová správně. Pomáhá uživateli pochopit, jak daný program funguje. Daný program je plně pod kontrolou ladicího nástroje, což vývojáři umožňuje sledovat tok programu a v jakýkoliv okamžik spuštěný program pozastavit a zkoumat stav programu a podle toho verifikovat jeho správnost. K zastavení běhu programu také slouží bod přerušení neboli breakpoint, což je místo, kde dojde k zastavení vykonávání programu. V tomto okamžiku ladicí nástroj notifikuje uživatele, kde a proč k zastavení došlo. Samotný uživatel může tyto body přerušení definovat pomocí rozhraní ladicího nástroje a tím i určovat ve který okamžik si přeje zkoumat stav programu.

Body přerušení lze rozdělit do dvou kategorií. První kategorií jsou instrukční breakpointy, které definují na kterém místě programu (instrukci) se má jeho vykonávání pozastavit. Obvykle bývají definovány konkrétní adresou instrukce, ale pro pohodlnost ovládání se definují pomocí pozice ve zdrojovém souboru daného programu, která se převede na konkrétní fyzickou adresu. Druhou kategorií jsou datové breakpointy (někdy též nazývané watchpointy). Tyto body přerušení pozastaví program tehdy, pokud je přistupováno do paměti na definovanou adresu anebo se do paměti zapisuje nebo čte definovaná hodnota. Ke každému bodu přerušení bývá obvykle možné definovat počet průchodů, kdy se daný bod přerušení ignoruje. Také je obvyklé definovat k bodu přerušení podmínku, jejímž splněním je přerušení aktivováno.

S body přerušení souvisí další nutná funkcionalita ladicího nástroje – krokování. Krokování umožnuje provést pouze jeden určitý krok programu a poté opět běh programu pozastavit a zkoumat jak se stav programu změnil. Při krokování může daný krok nabývat různých rozměrů, lze totiž krokovat po jednotlivých instrukcích nebo po jednotlivých taktech (pokud je model definován na úrovni taktu), nebo také po jednotlivých příkazech či funkčních voláních v případě, kdy je program napsán v některém z vyšších jazyků.

Pokud laděný program je pozastavený, vývojář pravděpodobně bude chtít zkoumat stav programu nebo případně jej i měnit. Znamená to tedy, že ladicí nástroj musí vývojáři umožnit číst a nastavovat obsahy všech registrů a hlavně číst a nastavovat obsah dat v paměti.

4.1 Uživatelské rozhraní ladicího programu v projektu Lissom

Uživatelské rozhraní ladicího programu v projektu Lissom je vestavěno do klientské aplikace prezentační vrstvy. Je tvořeno grafickým uživatelským rozhraním, které je součástí zásuvného modulu Eclissom pro platformu Eclipse. Další možnost použití ladicího programu je skrze textovou příkazovou řádku. V příkazové řádce je nutné se pro ladění programu přepnout do ladicího módu pomocí příkazu *gdb*, což naznačuje, že příkazy pro ladění se snaží být kompatibilní s ladicím nástrojem GDB [17], ale ve skutečnosti je k dispozici jen určitá podmnožina jeho příkazů.

Jedním z důležitých příkazů je příkaz *break*, pomocí něhož se dá nastavit instrukční breakpoint. Je podporováno vložení breakpointů na fyzickou adresu nebo je možné použít zdrojový

soubor a informaci o řádku, kde se má breakpoint vložit. Informace o převodu zdrojového souboru a řádku na adresu se načítají ze spustitelného souboru. Každému nově vloženému breakpointu je přiřazena číselná identifikace, pomocí které se s breakpointem manipuluje. Manipulací s breakpointy se rozumí smazání breakpointu pomocí příkazu *delete* nebo *clear*, povolování breakpointu pomocí příkazu *enable* a *disable*, nastavení počtu ignorovaných průchodů pomocí příkazu *ignore* a nastavení podmínky pomocí příkazu *condition*. Seznam breakpointů lze zobrazit pomocí příkazu *info breakpoints*.

Po zastavení běhu programu pomocí breakpointu lze programovat krokovat po instrukcích pomocí příkazu *step*. Opětovné rozběhnutí programu se provede příkazem *continue*.

Pro inspekci stavu procesoru lze použít hned několik příkazů. Prvním z nich je příkaz *print*, který vyhodnotí příkaz a jeho výsledek zobrazí. Existuje i podobný příkaz *set*, pomocí kterého lze nastavovat hodnoty, ale výsledek se nezobrazuje. Ve výrazech lze nejen přistupovat do paměti k proměnným pomocí názvu symbolů, ale také lze přistupovat k registrům pomocí znaku $ následovaného jménem registru. Obsah registrů lze zobrazit i použitím příkazu *info registers*.

Pro výrazy, které je možno použít v příkazech *set* a *print* nebo jako podmínku přiřadit k breakpointům, se používá syntaxe, která odpovídá podmnožině jazyka C. Ve výrazech lze přistupovat ke všem zdrojům procesoru a provádět s jejich hodnotami základní operace. Výsledek těchto operací je možné uložit pomocí operátoru přiřazení, ale jen na nejvyšší úrovni výrazu. Samotný překlad a vyhodnocení výrazu ovšem neprovádí příkazová řádka, ale je implementováno v rámci ladicí knihovny. Výrazy v breakpointech se vyhodnocují tak, že jakýkoliv výsledek různý od nuly je považován za kladný výsledek a breakpoint je povolen.

Ladicí režim příkazové řádky lze ukončit příkazem *exit*.

4.2 Ladicí knihovna

Jádro ladicího programu je implementováno v samostatné knihovně, která se jen připojuje k simulátoru při sestavování programu simulátoru. Simulátor pak s ladicí knihovnou komunikuje přes definované rozhraní.

Jako první věc po spuštění simulátoru dojde k inicializaci ladicí knihovny, při které se nastaví ukazatele pro přístup k paměti a zdrojům procesoru. V rámci inicializace se také vytvoří nové vlákno, které má na starost komunikaci simulátoru a ladicí knihovny se střední vrstvou, která do simulátoru přeposílá příkazy prezentační vrstvy a tedy i příkazy uživatelského rozhraní ladicího programu. Původní vlákno simulátoru pak obhospodařuje celou simulaci. Při inicializaci také dochází k načtení spustitelného souboru, který obsahuje kód, jenž se má odsimulovat. Simulátor kromě samotného kódu načítá ze spustitelného souboru další informace, zejména jde o tabulku symbolů a čísla řádků ve zdrojových souborech programu, které se používají pro podporu breakpointů.

Pro spuštění simulace je simulátoru zaslán příkaz *SML_DBL_START*, po jehož zpracování se simulace rozběhne. Při běhu simulace může dojít k přerušení simulace, pokud program narazil na breakpoint. Samotná kontrola breakpointu, tedy zda se má simulace zastavit, je implementována ve funkci ladicí knihovny, která se volá před každým dekódováním instrukce a zjišťuje, zda je na dané adrese nastavený breakpoint. V rámci této funkce se zavolá metoda, která se pokusí k dané adrese najít breakpoint a případně vyhodnotit jeho podmínku nebo zjistit jestli se nemá daný breakpoint ignorovat. Pokud dojde k zastavení simulace, simulátor odešle zprávu střední vrstvě informující o místě, kde k zastavení došlo. Zpráva je pak přeposlána prezentační vrstvě a informace o zastavení běhu simulace je zobrazena uživateli.

Pokud simulace neběží (byla zastavena pomocí breakpointu), lze simulaci krokovat. Při krokování se simulace vždy znovu spouští, jen je před každým dekódováním instrukce proveden test, který kontroluje, zda je simulace v krokovacím módu. Pokud ano, je simulace zastavena podobným způsobem jako při dosažení breakpointu.

5 Standard Nexus 5001

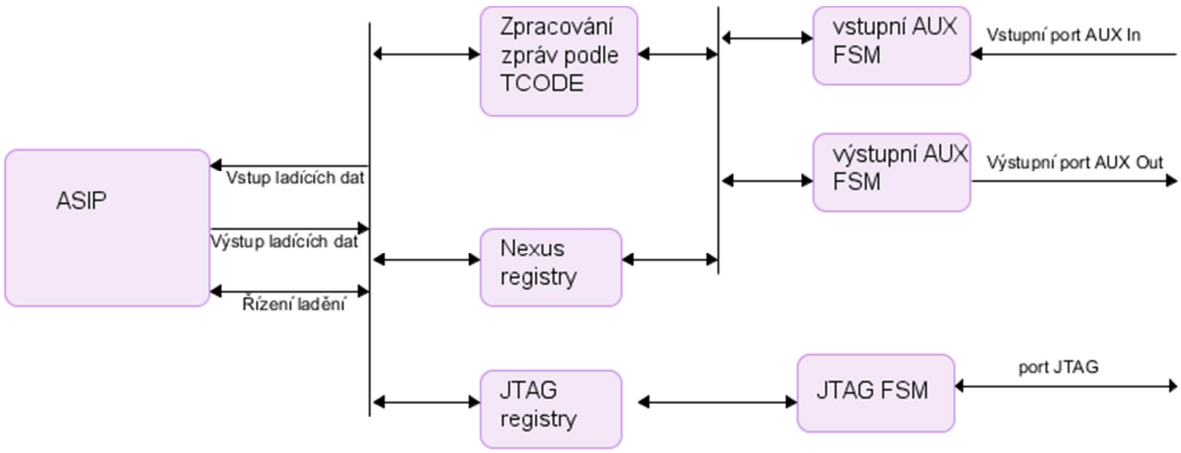
Standard IEEE-ISTO 5001-2003 [2] (dále nazývaný stručně Nexus) je standard pro ladicí rozhraní vestavěných systémů, který byl vytvořen organizací The Nexus 5001 Forum. Cílem této organizace bylo vytvoření standardu, který bude obsahovat funkčně bohaté ladicí rozhraní, které bude zároveň nezávislé na daném procesoru a architektuře. Toto nezávislé rozhraní je nutné k tomu, aby výrobci ladicích nástrojů mohli vytvářet své produkty s konzistentní funkcionalitou pro široké spektrum procesorů. Tím dojde zároveň ke snížení nákladů, které jsou spojeny s vývojem ladicích nástrojů vytvořených pro jeden konkrétní procesor a architekturu.

5.1 Architektura rozhraní Nexus

Standard Nexus mimo jiné adresuje i problém minimalizace počtu pinů ladicího rozhraní daného čipu. Pokud je kritické mít co nejmenší počet pinů je možné pro pokrytí základní funkcionality použít JTAG [18] port. Pro podporu ostatních tříd definuje standard přídavný port Nexus Auxiliary Port, který doplňuje JTAG port nejmenším možným počtem bitů, čehož je dosaženo i tím, že datový výstup zpráv MDO (message data out) může mít variabilní bitovou šířku, podle toho jak velký datový tok je požadován.

Rozšiřující AUX port má také výhodu pro použití s multiprocesorovými systémy, při jejichž ladění je nutné přenášet větší objemy dat než při ladění jednoprocesorového systému. Tyto data obsahují nejen informace o laděných programech pro jednotlivé procesory, ale také je nutné sledovat komunikaci mezi procesory a přístupy ke sdíleným zdrojům.

Samotná komunikace přes ladicí port pak probíhá pomocí zpráv. Jednotlivé zprávy jsou samostatné a obsahují zdroj i cíl dané zprávy (dané jádro procesoru a daný registr), typ informace a samotný obsah zprávy závislý na typu zprávy. Celá struktura zprávy je definována pomocí hlavičky TCODE.



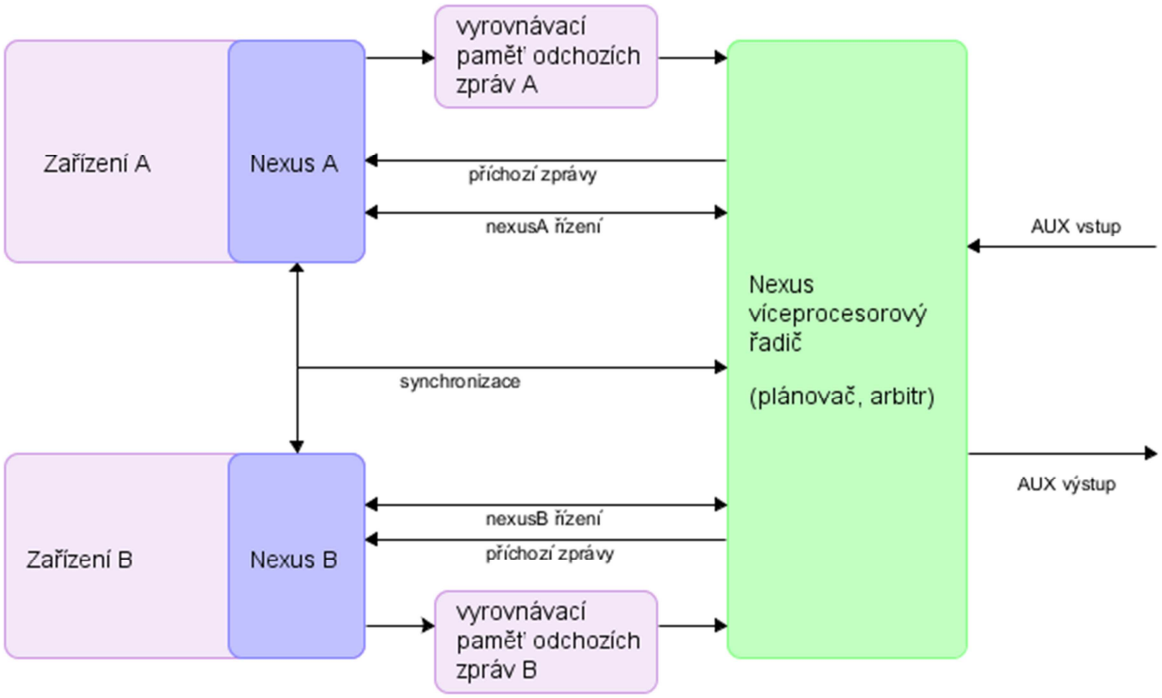
Obrázek 5.5.1: Architektura ladicího rozhraní Nexus

5.2 Ladění víceprocesorových systémů pomocí rozhraní Nexus

Jak již bylo zmíněno v předchozí kapitole, ladění víceprocesorových systémů vyžaduje více ladicích informací, zejména informace o meziprocesorové komunikaci. Standard Nexus umožňuje ladění až 32 zařízení v rámci systému. Každé takovéto zařízení, které umožňuje ladění, se pak nazývá klient.

Vzhledem k tomu, že komunikace definovaná ve standardu Nexus je založena na zprávách, je nutné nějakým způsobem identifikovat zařízení, kterému je daná zpráva určena, případně které danou zprávu odeslalo. K tomuto slouží registr identity zařízení (Device Identity Register – DID), který obsahuje identifikaci daného zařízení. Pokud chceme přistupovat k zařízení, například přečíst hodnotu registru, je nutné nastavit registr pro výběr klienta (Client Select Control - CSC). Po nastavení registru CSC jsou zprávy doručovány danému klientovi.

Samotné posílání zpráv z ladicího nástroje je jednoduché, v jeden okamžik může být odeslána jen jedna zpráva a ta může být hned dekódována a doručena danému zařízení. Ovšem u zpráv odesílaných ze zařízení do ladicího nástroje mohou vznikat problémy. Jedním z nich je, že dané zařízení bude generovat zprávy větším množství, než je datová propustnost Nexus portu. Tento problém je ještě znásoben, pokud tyto zprávy generuje více zařízení najednou. Pro tyto účely je vhodné mít dostatečně velkou vyrovnávací paměť pro odesílané zprávy, aby se zamezilo ztrátám těchto zpráv. Pokud se více zařízení dožaduje přístupu k AUX portu pro odeslání zprávy, lze tento problém řešit několika způsoby. Nejjednodušší je, že zprávy od zařízení, které není vybráno pomocí CSC, nebudou odesílány. Nebo lze implementovat složitější řízení odesílání zpráv podle priorit.



Obrázek 5.5.2: Použití ladicího rozhraní Nexus ve víceprocesorovém systému

5.3 Funkce ladicího rozhraní Nexus

Standard Nexus 5001 Forum nespecifikuje jedno konkrétní rozhraní, které je nutné vždy implementovat, ale rozděluje funkce do několika tříd. V současné době definuje čtyři třídy vlastností vestavěných procesorů, přičemž Třída 1 poskytuje jen minimální ladicí rozhraní a na druhé straně Třída 4 poskytuje kompletní sadu programových funkcí.

Třida 1 podporuje ovládání běhu – spuštění, zastavení, čtení/zápis do paměti když procesor neběží, instrukční i datové breakpointy (watchpointy), čtení a nastavování registrů

Třída 2 přidává sledování vlastnictví a sledování toku programu. Sledování vlastnictví umožnuje sledování aktuální úlohy nebo aktuálního procesu pro systémy postavené na real-time jádrech nebo real-time operačních systémech.

Třída 3 přidává sledování zápisu do paměti a také možnost číst a zapisovat do paměti bez zastavení procesoru.

Třída 4 přidává nahrazování paměti, které umožní načítání instrukcí a dat přes AUX port

5.3.1 Funkce Třídy 1 a porovnání s ladicím nástrojem Lissom

V této podkapitole jsou popsány jednotlivé ladicí funkce, které bylo nutné implementovat pro Třídu 1. Pro každou funkci je zde i napsáno, která funkce z Nexus API slouží pro provedení dané ladicí operace. Dále je u každé funkce napsán ekvivalentní příkaz pro ladicí příkazovou řádku z projektu Lissom a také příslušný příkaz pro střední vrstvu.

5.3.1.1 Čtení a zápis registrů v ladicím módu

V Nexus API se provede voláním nx_Control()
V příkazové řádce se dá provést použitím příkazů *print* a *set* které pracují s výrazy – k registrům se přistupuje pomocí znaku $ následovaného jménem registru. Také lze vypsat seznam registrů a jejich hodnot pomocí příkazu *info registers*.
Příkaz pro střední vrstvu DBL_EXPRESSION

5.3.1.2 Čtení a zápis paměti v ladicím módu

V Nexus API se provede voláním nx_WriteMem() a nx_ReadMem()
V příkazové řádce se zapisuje do paměti pomocí provedení výrazů – příkazy *print* a *set*
Příkaz pro střední vrstvu DBL_EXPRESSION

5.3.1.3 Vstup do ladicího módu pomocí resetu

V Nexus API se provede voláním nx_Control()
Tato funkcionalita je prakticky stejná, jako kdyby se nastavil breakpoint na první instrukci, která se má provést. V příkazové řádce se spustí simulace v ladicím módu pomocí nově přidaného příkazu *start*.
Byl přidán nový příkaz pro střední vrstvu DBL_DEBUG_ALL

5.3.1.4 Vstup do ladicího módu z uživatelského módu

V Nexus API se provede voláním nx_Control()
Tato funkcionalita spočívá v tom, že uživatel je kdykoliv schopen zastavit procesor a přejít do ladicího módu, kde bude moci provádět všechny podporované operace (číst/zapisovat registry atd.). Do příkazové řádky byl dolněn nový příkaz *stop*.
Byl přidán nový příkaz pro střední vrstvu DBL_STOP_ALL

5.3.1.5 Ukončení ladicího módu a přechod do uživatelského módu

V Nexus API se provede voláním nx_Control()
Pokračování simulace lze provést provedením příkazu *continue*.
Příkaz pro střední vrstvu DBL_RESUME_ALL

5.3.1.6 Provedení jedné instrukce v uživatelském módu a přechod do ladicího módu

V Nexus API se provede voláním nx_SetEvent()
Provedení pouze jedné instrukce lze provést pomocí příkazu *step*
Příkaz pro střední vrstvu DBL_STEP

5.3.1.7 Zastavení provádění programu pomocí instrukčního breakpointu nebo datového breakpointu a přechod do ladicího módu

V Nexus API se provede voláním nx_SetEvent()
V Nexus standardu se používají tři označení pro breakpointy. První z nich je Instrukční breakpoint - zastaví procesor na předdefinované instrukci (adrese). Druhý je datový breakpoint (v jiné literatuře někdy označován jako watchpoint) - zastaví procesor, pokud je přistup do paměti nebo hodnota dat rovna předdefinované adrese nebo hodnotě. Třetí termín je watchpoint – je to to samé jako breakpoint (instrukční i datový), ale nezastaví procesor - jen se signalizuje, že daná podmínka byla splněna. Původní ladicí nástroj projektu Lissom podporoval jen instrukční breakpointy, podpora pro datové breakpointy byla přidána v rámci této práce. Rovněž watchpointy nebyly podporovány, ale vzhledem k tomu, že breakpointy lze ignorovat, stačila jen malá úprava, aby ladicí nástroj posílal notifikace o tom, že prošel takovýmto breakpointem.

5.3.1.8 Možnost nastavit breakpoint nebo watchpoint

V Nexus API se provede voláním nx_SetEvent()
Instrukční breakpoint lze nastavit v příkazové řádce pomocí příkazu *break*. Byly přidány také nové příkazy pro podporu datových breakpontů *watch*, *rwatch* a *awatch*.
Příkaz pro střední vrstvu DBL_BREAKPOINT a DBL_WATCHPOINT

5.3.1.9 Identifikace zařízení

V Nexus API se provede voláním nx_Control()
Není podporováno v ladicím nástroji projektu Lissom

5.3.1.10 Možnost odeslat zprávu, pokud je podmínka watchpointu splněna.

V Nexus API se provede voláním nx_SetEvent()
Tato funkcionalita byla implementována pomocí vypnutých breakpointů, kdy pokud dojde k zásahu vypnutého breakpointu, je zaslána notifikační zpráva.

5.3.2 Funkce Třídy 2

5.3.2.1 Sledování vlastnictví

Sledování vlastnictví procesoru umožňuje vytvořit pohled běh na úrovní abstrakce jednotlivých úloh. Nabízí tak nejvyšší míru abstrakce pro sledování běhu operačního systému a je vhodné zejména v případě když vývojář nemá zájem o informace na nižších úrovních abstrakce. Sledování vlastnictví je zejména důležité pro vestavěné procesory s jednotkou pro správu paměti, ve kterých všechny programy používají stejný programový i datový logický adresový prostor a umožňuje tak ladicímu nástroji určit, která část symbolických instrukcí a které zdrojové soubory jsou nutné pro ladění na nižší úrovni abstrakce.

Sledování vlastnictví je ve standardu Nexus definováno jako posílání zpráv OTM (Ownership Trace Messaging) přes AUX port. Každá taková zpráva obsahuje informace o tom, který proces (jeho ID) nebo která úloha operačního systému je aktivní. Tato zpráva je vždy odeslána, pokud je proces nebo úloha aktivována. Ve standardu Nexus je definován registr OTM, do kterého se zapisuje ID procesu nebo úlohy. Pokud procesor obsah tohoto registru aktualizuje, je odeslána zpráva přes AUX port.

5.3.2.2 Sledování toku programu

Sledování toku programu umožňuje sledovat všechny změny v toku programu, tedy všechny přímé i nepřímé skoky a výjimky. Ve standardu Nexus je definováno posílání zpráv BTM (Branch Trace Messaging), které obsahují informace o počtu instrukcí, které byly vykonány od posledního skoku. Pro nepřímé skoky se posílá také informace o adrese cíle skoku.

Při sledování toku programu může dojít k chybě a to zejména v tom případě, pokud množství zpráv pro sledování toku programu překročí datovou propustnost AUX portu. V takovém případě je nutné implementovat speciální zprávu, která informuje vývojáře, že informace o sledování byla ztracena.

Na některých architekturách procesorů, jako například procesory s redukovanou instrukční sadou (RISC), mohou běžet aplikace, které za běhu provedou velké množství přímých skoků a minimum skoků nepřímých. Zprávy o přímých skocích ovšem neobsahují adresu o cíli skoku a proto ladicí nástroj tyto zprávy musí vyhodnocovat pouze relativně podle počtu vykonaných instrukcí. Z tohoto důvodu je definována speciální zpráva pro synchronizaci sledování toku programu, která zaručí synchronizaci ladicího nástroje s daným tokem. Tato zpráva je poslána po resetu procesoru a pak periodicky po odeslání nejvíce 256 BTM zpráv.

5.3.3 Funkce Třídy 3

5.3.3.1 Sledování dat

Procesory třídy 3 musí implementovat sledování zápisu dat. Sledování čtení může být implementováno, ale je nepovinné. Pro sledování paměti musí být možno nastavit počáteční a koncovou adresu, pro kterou má být sledování prováděno. Datový přístup do této paměti je pak monitorován a v případě přístupu do této paměti je odeslána zpráva DTM (Data Trace Messaging) přes AUX port. Tato zpráva obsahuje data a unikátní část adresy. Celá část adresy není uvedena, aby se snížil objem dat odesílaný přes AUX port. Celá adresa se pak sestavuje relativně na základě předchozích DTM zpráv. Z důvodu, že počet DTM zpráv může být příliš velký a můžou se některé

zprávy ztratit, musí být implementováno zaslání zprávy o přetečení fronty a ztracení zprávy o sledování paměti. Z tohoto důvodu se také pravidelně posílají zprávy obsahující kompletní adresu zaručující plnou synchronizaci ladicího nástroje a laděného zařízení. Tyto zprávy jsou zasílány po inicializaci procesoru a poté pravidelně po odeslání nejvíce 256 DTM zpráv, podobně jako synchronizační zpráva pro sledování programu.

5.3.3.2 Čtení a zápis paměti

Čtení a zápis paměti vestavěných procesorů třídy 3 umožňuje přístup k paměti a zdrojům mapovaným do paměti. Čtení a zápis do pamětí musí být možný, když procesor je v zastaveném stavu a také i ve stavu, když procesor běží. Tato funkce může být implementována buď pomocí zpráv zasílaných přes AUX port, nebo pomocí přístupu přes JTAG port. V rámci simulátoru v projektu Lissom je již tato funkcionalita podporována pomocí příkazů *set* a *write*, které je možno spustit i když běží simulace.

5.3.4 Funkce Třídy 4

5.3.4.1 Nahrazování paměti

Vestavěné procesory třídy 4 musí podporovat možnost aktivování nahrazení paměti přes AUX port. Nahrazení paměti musí být možné aktivovat po resetu, ale je možné implementovat nepovinně aktivování nahrazení paměti při aktivování watchpointu nebo při datovém přístupu nebo načtení instrukce z definovaného adresového rozsahu.

Pokud je nahrazování paměti aktivováno, procesor čte instrukce i data přes AUX port z ladicího nástroje.

5.4 Zprávy zasílané přes Nexus AUX port

AUX port poskytuje vysokorychlostní komunikační kanál pro komunikaci mezi ladicím nástrojem a laděným procesorem. Veškerá komunikace proudící přes tento kanál oběma směry má formát zpráv.

Formát a význam některých zpráv je definován ve standardu Nexus[2]. Ostatní zprávy můžou být definovány výrobcem procesorů. V tomto případě ale i ladicí nástroj musí obsahovat rozšířenou funkcionalitu, aby tyto zprávy mohl zpracovat.

Každá zpráva obsahuje na začátku 6-bitovou hodnotu TCODE, která definuje typ zprávy. Hodnoty 0 až 55 indikují, že zpráva je takzvaná veřejná zpráva (public message), která je definována ve standardu Nexus. Hodnoty 56 až 62 jsou určeny pro zprávy definované výrobcem. Hodnota 63 také indikuje zprávu definovanou výrobcem, ale určuje, že přesný typ zprávy se určí až podle následujících druhotných informací.

Jednotlivé zprávy jsou dále děleny na pakety, které obsahují jednotlivé části informace přenášené ve zprávě. Velikost těchto paketů může být pevná nebo variabilní. Některé pakety ve veřejných zprávách standardu Nexus jsou označeny jako *vendor-fixed*, což znamená, že tyto packety mají pevnou délku, ale je umožněno výrobci tyto pakety neimplementovat – potom bude mít tento paket nulovou délku. Ladicí nástroj pak musí podle identifikace zařízení zjistit, zda tyto pakety jsou ve zprávách obsaženy nebo ne. Podobně je umožněno vynechat i pakety označené jako *vendor-variable*.

Veřejné zprávy se dají shrnout do několika základních kategorií:

- Zpráva Debug status - je zaslána, pokud v procesoru dojde ke změně stavu ladění, např. při vyjímce, nebo při detekci breakpoint
- Zpráva device ID – zaslána procesorem, obsahuje identifikaci daného procesoru
- Skupina zpráv pro sledování vlastnistvíí
- Skupina zpráv pro sledování dat
- Skupina zpráv pro sledování toku programu
- Chybová zpráva Error
- Zpráva watchpoint match – notifikace při zásahu watchpointu
- Skupina zpráv pro přístup k doporučeným registrům nexus
- Skupina zpráv pro přístup k paměti
- Skupina zpráv pro nahrazení portů

Mezi těmito zprávami chybí zprávy pro nastavení breakpointů, krokování a základní ladicí funkce. Standard Nexus totiž počítá, že všechny tyto ladicí informace se čtou nebo nastavují do registrů, které standard označuje jako Doporučené registry Nexus. Jedním z těchto registrů je i Developlment control register, který mimo jiné slouží k povolení ladicího módu, k nastavení požadavku na vstup do ladicího módu nebo k povolení krokování. Druhým důležitým registrem je Development status register, který obsahuje informace o důvodu, proč byl procesor přepnut do ladicího módu. Obsahuje tedy informace o chybách, stavu procesoru a informace zda došlo k zasažení některého z breakpointů. Samotné nastavení breakpointů probíhá pomocí nastavení dalších speciálních registrů.

6 Návrh a implementace ladicího nástroje pro víceprocesorový systém na čipu

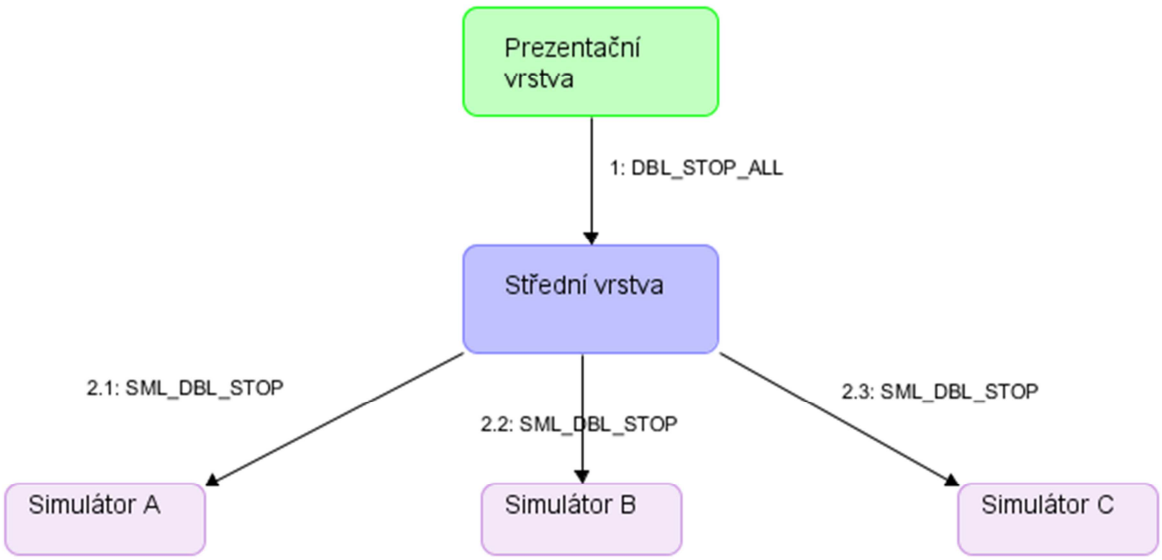
6.1 Návrh a implementace chybějící funkcionality třídy 1 standardu Nexus

Z výše uvedeného porovnání funkcionality ladicího nástroje Lissom a standardu Nexus vyplývá, že do ladicího nástroje bylo třeba doplnit následující funkcionalitu třídy 1: Vstup do ladicího módu pomocí resetu, vstup do ladicího módu z uživatelského módu, datové breakpointy a notifikace pro watchpointy (deaktivovaných breakpointy).

6.1.1 Funkce pro vstup do ladicího módu

Funkce pro vstup do ladicího módu pomocí resetu byla implementována poměrně jednoduše. Do příkazové řádky byl přidán nový příkaz *start*, který způsobí rozeslání nového příkazu SML_DBL_START ze střední vrstvy všem simulátorům. Po obdržení tohoto příkazu se simulátor spustí a ihned se přepne do krokovacího módu.

Funkce pro vstup do ladicího módu z uživatelského módu je jen o něco málo složitější. Byl přidán nový příkaz *stop* do příkazové řádky, který zaručí rozeslání příkazu SML_DBL_STOP pomocí střední vrstvy k simulátorům. V samotném simulátoru se tento příkaz obslouží jednoduše tím, že se opět nastaví krokovací mód simulátoru.



Obrázek 6.1: Schéma komunikace při běhu víceprocesorové simulace v případě, kdy dojde k zadání příkazu *stop* v příkazové řádce.

6.1.2 Watchpointy

Samotná funkcionalita watchpointů byla implementována pomocí použití deaktivovaných breakpointů. Do funkce pro testování, zda se na dané adrese nachází breakpoint, byla doplněna funkce, která zašle danou notifikační zprávu SML_DBL_STATE_BPNOTIFY, pokud se na dané adrese program nemá zastavit, ale je na ní nastavený deaktivovaný breakpoint. V případě datového braekpointu se zasílá zpráva SML_DBL_STATE_WPNOTIFY. Tato zpráva je pak přeposlána přes střední vrstvu až do příkazové řádky, která ji prezentuje uživateli.

6.1.3 Identifikace zařízení

Funkce pro identifikaci zařízení nemá z pohledu simulátoru význam, a proto nebyla implementována. Popis identifikace simulátorů (a tím i zařízení) bude popsán v následující kapitole, která se zaobírá podporou pro ladění víceprocesorových systémů. Případná implementace by ovšem byla docela snadná. Pro tuto funkci je opět nutné přidat nový příkaz do příkazové řádky a zaručit přeposlání příkazu přes střední vrstvu do simulátoru. Ovšem nejdůležitější je pro tuto funkci doplnit do simulátoru registr, který bude obsahovat danou identifikace zařízení. Tuto hodnotu bude číst pouze ladicí nástroj. Zápis do toho registru nebude povolen, respektive při pokusu zapsat do tohoto registru se nic nestane, jeho hodnota zůstane zachována.

6.1.4 Datové breakpointy

Z pohledu implementace změn pro podporu třídy 1 standardu Nexus bylo ve stávajícím ladicím programu nejsložitější implementovat podporu pro datové breakpointy. Bylo nutné implementovat změny v příkazové řádce, ve střední vrstvě a generátoru nástrojů a také i v samotné ladicí knihovně.

6.1.4.1 Datové breakpointy – příkazová řádka

Do příkazové řádky byly doplněny nové příkazy *watch* (pro sledování zápisu na danou adresu), *rwatch* (pro sledování čtení z dané adresy) a *awatch* (pro sledování čtení i zápisu), které jsou zvoleny podle stejných funkcí v ladicím nástroji GDB. V ladicím nástroji GDB se datové breakpointy nazývají *watchpointy*, proto i v ladicím nástroji projektu Lissom se v uživatelském rozhraní a i v názvech tříd použitých pro označení datových breakpointů používá výraz *watchpoint*. Pro účely této práce ale bude používán nadále výraz datový breakpoint tak, jak je definován ve standardu Nexus.

Všechny tři tyto nové příkazy mají stejnou syntaxi, která obsahuje jeden povinný a jeden nepovinný parametr. Povinný parametr určuje adresu v paměti, kterou se má sledovat. Je možné použít buď přímo fyzickou adresu, nebo lze také použít název symbolu, jehož adresa se pak dohledá pomocí tabulky symbolů, která se vytvoří na základě informací ve spustitelném souboru. Místo adresy lze také použít výraz *ANY*, který specifikuje, že se mají hlídat všechny přístupy do paměti. Druhý nepovinný parametr označuje hodnotu watchpointu, tedy jaká hodnota se zapsala nebo přečetla z paměti. Pokud není tento druhý parametr uveden, watchpoint zastaví program při každém přístupu na danou adresu.

Pro úpravu datových breakpointů se používají stejné příkazy jako pro úpravu vlastností standardních instrukčních breakpointů. Je to implementováno z důvodu kompatibility s ladicím nástrojem GDB. Pro povolení nebo zakázání datového breakpointu se použije příkaz *enable* a pro smazání datového breakpointu příkaz *delete*. Lze také použít příkaz *ignore* na nastavení počtu

ignorovaných zásahů datového breakpointu nebo příkaz *condition* pro nastavení podmínky datového breakpointu.

K vypsání seznamu všech datových breakpointů slouží příkaz *info breakpoints* – tedy dojde k vypsání seznamu breakpointů, ve kterém je u každého breakpointu uveden jeho typ – tedy jestli je datový (*watchpoint*) nebo instrukční (*breakpoint*).

6.1.4.2 Úprava simulátoru

Pro podporu datových breakpointů bylo potřeba upravit simulátor, respektive upravit generátor simulátoru, který vygeneruje simulátor daného procesoru podle popisu a parametrů. Hlavní úpravou simulátoru je přidání možnosti sledování přístupu do paměti, při kterém dochází ke čtení a zápisu dat, aby bylo možné tyto hodnoty porovnat s předdefinovanými hodnotami datového breakpointu a v případě shody zastavit program.

Čtení a zápis do paměti je implementován v generovaných inline funkcích, které simulují čtení a zápis do paměti. Tyto funkce jsou generovány podle parametrů paměti, které jsou definovány v modelu popsaném v jazyce ISAC, mezi něž patří význam pořadí bytů ve slově a bitová šířka slova. Funkce jsou generovány ve variantách jak pro zarovnaný, tak i pro nezarovnaný přístup do paměti. V generovaných funkcích se zapisují sledovaná data do globálních proměnných. Mezi sledovaná data patří typ přístupu do paměti (čtení nebo zápis), adresa odkud se četlo nebo kam se zapisovalo a také samotná data, jež se četly nebo zapsaly. Jelikož v rámci jedné instrukce může dojít k obecně více přístupům do paměti, je nutné, aby tyto globální proměnné byly typu pole, aby bylo možné do nich zaznamenat informace o všech přístupech do paměti. Generované funkce pro čtení a zápis paměti jsou pomocí zjednodušeného pseudokódu znázorněny na obrázku 6.2 a 6.3. Tyto funkce jsou pro přístup do paměti při čtení nebo zápisu celého bloku dat a je uvažována architektura, kde slovo (blok) má šířku 32bitů a tvoří ho 4 8bitové subbloky. Proměnná *memory* je pole reprezentující paměť v simulátoru a obsahuje jednotlivé bloky jako prvky pole. Pro potřeby testování aktivace datového breakpointu se ukládá adresa do pole DBG_WATCHPOINTS_ADDRS, typ přístupu čtení do pole DBG_WATCHPOINTS_TYPES a čtená hodnota do pole DBG_WATCHPOINTS_VALUES. Bitové operace nad daty a dvěma bloky pro podporu nezarovnaného přístupu do paměti jsou v pseudokódu nahrazeny funkci *unaligned_read* respektive *unaligned_write*.

```
ulong memory_read(ulong ind)
{
DBG_WATCHPOINTS_ADDRS[DBG_WATCHPOINTS_IDX + 0] = ind + 0;
DBG_WATCHPOINTS_TYPES[DBG_WATCHPOINTS_IDX + 0] = TYPE_READ;
DBG_WATCHPOINTS_ADDRS[DBG_WATCHPOINTS_IDX + 1] = ind + 1;
DBG_WATCHPOINTS_TYPES[DBG_WATCHPOINTS_IDX + 1] = TYPE_READ;
DBG_WATCHPOINTS_ADDRS[DBG_WATCHPOINTS_IDX + 2] = ind + 2;
DBG_WATCHPOINTS_TYPES[DBG_WATCHPOINTS_IDX + 2] = TYPE_READ;
DBG_WATCHPOINTS_ADDRS[DBG_WATCHPOINTS_IDX + 3] = ind + 3;
DBG_WATCHPOINTS_TYPES[DBG_WATCHPOINTS_IDX + 3] = TYPE_READ;

ulong rdata = 0;
switch(ind % 4) {
  case 0: // zarovnaný přístup do paměti
    rdata = memory[(ind / 4)];
    break;
```

```
  case 1: // nezarovnaný přístup do paměti, data se čtou ze dvou
  case 2: // bloků a pomocí bitových operátorů je přečtena požadovaná
  case 3: // hodnota, zde znázorněno pomocí funkce unaligned_read
    rdata = unaligned_read(memory [(ind/4)], memory [(ind/4)+1],ind%4);
    break;
  }
DBG_WATCHPOINTS_VALUES[DBG_WATCHPOINTS_IDX + 0] = rdata;
DBG_WATCHPOINTS_VALUES[DBG_WATCHPOINTS_IDX + 1] = rdata;
DBG_WATCHPOINTS_VALUES[DBG_WATCHPOINTS_IDX + 2] = rdata;
DBG_WATCHPOINTS_VALUES[DBG_WATCHPOINTS_IDX + 3] = rdata;
DBG_WATCHPOINTS_IDX+=4;
return rdata;
}
```

Obrázek 6.2: zjednodušená generovaná funkce pro čtení paměti v pseudokódu. Adresa, odkud se čtou data, je obsažená v parametru ind.

```
void memory_write(ulong ind, ulong data)
{
DBG_WATCHPOINTS_ADDRS[DBG_WATCHPOINTS_IDX + 0] = ind + 0;
DBG_WATCHPOINTS_TYPES[DBG_WATCHPOINTS_IDX + 0] = TYPE_WRITE;
DBG_WATCHPOINTS_VALUES[DBG_WATCHPOINTS_IDX + 0] = data;
DBG_WATCHPOINTS_ADDRS[DBG_WATCHPOINTS_IDX + 1] = ind + 1;
DBG_WATCHPOINTS_TYPES[DBG_WATCHPOINTS_IDX + 1] = TYPE_WRITE;
DBG_WATCHPOINTS_VALUES[DBG_WATCHPOINTS_IDX + 0] = data;
DBG_WATCHPOINTS_ADDRS[DBG_WATCHPOINTS_IDX + 2] = ind + 2;
DBG_WATCHPOINTS_TYPES[DBG_WATCHPOINTS_IDX + 2] = TYPE_WRITE;
DBG_WATCHPOINTS_VALUES[DBG_WATCHPOINTS_IDX + 0] = data;
DBG_WATCHPOINTS_ADDRS[DBG_WATCHPOINTS_IDX + 3] = ind + 3;
DBG_WATCHPOINTS_TYPES[DBG_WATCHPOINTS_IDX + 3] = TYPE_WRITE;
DBG_WATCHPOINTS_VALUES[DBG_WATCHPOINTS_IDX + 0] = data;
DBG_WATCHPOINTS_IDX+=4;

switch(ind % 4) {
  case 0: // zarovnaný přístup do paměti
    memory [(ind  / 4)] = data;
    return;
  case 1: // nezarovnaný přístup do paměti, data se čtou ze dvou
  case 2: // bloků a pomocí bitových operátorů je zapsána požadovaná
  case 3: // hodnota, zde znázorněno pomocí funkce unaligned_write
    unaligned_write(memory[(ind/4)], memory[(ind/4)+1], ind % 4, data);
    return;
  }
}
```

Obrázek 6.3: zjednodušená generovaná funkce pro zápis dat do paměti v pseudokódu. Adresa, kam se mají data zapsat, je obsažená v parametru ind.

V generovaných funkcích je ošetřeno sledování přístupu do paměti i na nezarovnané adressy. Datový breakpoint může být nastaven na zarovnanou i na nezarovnanou adresu, ale k zásahu daného datového breakpointu dojde jen tehdy, pokud byl přístup přímo na první subblok, na jehož adresu je nastaven breakpoint.

Detailněji je způsob aktivace datového breakpointu při nezarovnaném přístupu zachycen v následujícím příkladě, při kterém je použita architektura, kde slovo (blok) má šířku 32bitů a tvoří ho 4 8bitové subbloky. Pokud se datový breakpoint nastaví na adresu 4 a dojde k zápisu celého 32bitového slova na adresu 2, je datový breakpoint aktivován. Pokud se ale na adresu 2 zapíše jen 16bitů, datový breakpoint aktivován není. K aktivaci nedojde ani v případě pokud se zapíše např. 16 bitů na adresu 5. Detailněji jsou tyto případy zobrazeny na obrázcích 6.4, 6.5 a 6.6, kde je zvýrazněna adresa ta, na které je nastavený breakpoint, a zvýrazněné data jsou ty, k jejich zápisu došlo do paměti na danou adresu.

```
    +0   +1   +2   +3
0:  0x01 0x23 0x45 0x67
4:  0x89 0xab 0xcd 0xef
```

Obrázek 6.4: Schéma nezarovnaného přístupu do paměti, datový breakpoint nastavený na adresu 4 je aktivován zápisem 32 bitů na adresu 2

```
    +0   +1   +2   +3
0:  0x01 0x23 0x45 0x67
4:  0x89 0xab 0xcd 0xef
```

Obrázek 6.5: Schéma nezarovnaného přístupu do paměti, datový breakpoint nastavený na adresu 4 není aktivován zápisem 16 bitů na adresu 2

```
    +0   +1   +2   +3
0:  0x01 0x23 0x45 0x67
4:  0x89 0xab 0xcd 0xef
```

Obrázek 6.6: Schéma nezarovnaného přístupu do paměti, datový breakpoint nastavený na adresu 4 není aktivován zápisem 16 bitů na adresu 5

Samotné testování, zda se došlo k zásahu datového breakpointu, čili zda se má program zastavit, je implementováno v rámci řízení vykonávání programu - k zastavení tedy dojde před načtením následující instrukce. Test se provádí pomocí volání funkce ladicí knihovny pro každý zaznamenaný přístup do paměti během vykonávání předchozí instrukce. Tato funkce pak testuje, zda daný přístup do paměti má způsobit pozastavení vykonávání programu.

Protože sledování paměti a následné testování těchto dat na možnost zásahu datového breakpointu může znatelně zpomalit simulaci, byla přidána parametr do generátoru nástrojů, který povoluje možnost použití datových breakpointů v generovaném kódu. V případě, kdy není plánováno použití datových breakpointů pro ladění aplikace, lze simulátor vygenerovat bez této podpory a simulace programu pak bude rychlejší. Kromě samotného kódu pro testování zásahu datového breakpointu, který se provádí před simulací provádění následující instrukce, zpomaluje simulaci také kód, který zaznamenává přístupy do paměti. Na obrázcích 6.2 a 6.3 jsou vygenerované funkce s podporou datových breakpointů, stejné vygenerované funkce bez této podpory jsou v pseudokódu na následujících obrázcích 6.7 a 6.8. Porovnáním těchto pseudokódů je zjevná redukce prováděných operací, která se projeví v rychlosti simulace.

```
ulong memory_read(ulong ind)
{
switch(ind % 4) {
  case 0: // zarovnaný přístup do paměti
    return = memory[(ind / 4)];
  case 1: // nezarovnaný přístup do paměti, data se čtou ze dvou
  case 2: // bloků a pomocí bitových operátorů je přečtena požadovaná
  case 3: // hodnota, zde znázorněno pomocí funkce unaligned_read
    return unaligned_read(memory [(ind/4)], memory [(ind/4)+1],ind%4);
  }
}
```

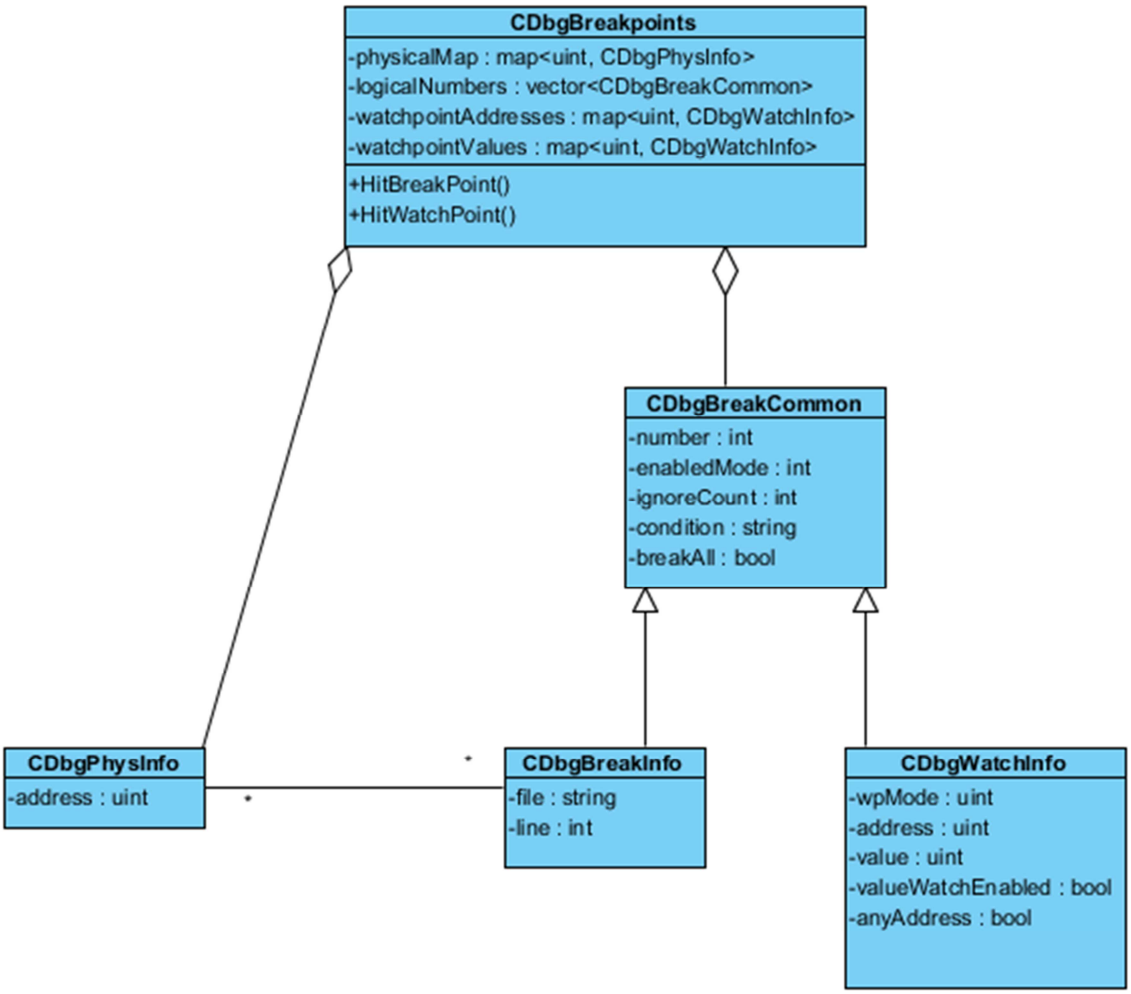
Obrázek 6.7: vygenerovaná funkce simulující čtení z paměti v pseudokódu bez podpory datových breakpointů.

```
void memory_write(ulong ind, ulong data)
{
switch(ind % 4) {
  case 0: // zarovnaný přístup do paměti
    memory [(ind  / 4)] = data;
    return;
  case 1: // nezarovnaný přístup do paměti, data se čtou ze dvou
  case 2: // bloků a pomocí bitových operátorů je zapsána požadovaná
  case 3: // hodnota, zde znázorněno pomocí funkce unaligned_write
    unaligned_write(memory[(ind/4)], memory[(ind/4)+1], ind % 4, data);
    return;
  }
}
```

Obrázek 6.8: vygenerovaná funkce simulující zápis do paměti v pseudokódu bez podpory datových breakpointů.

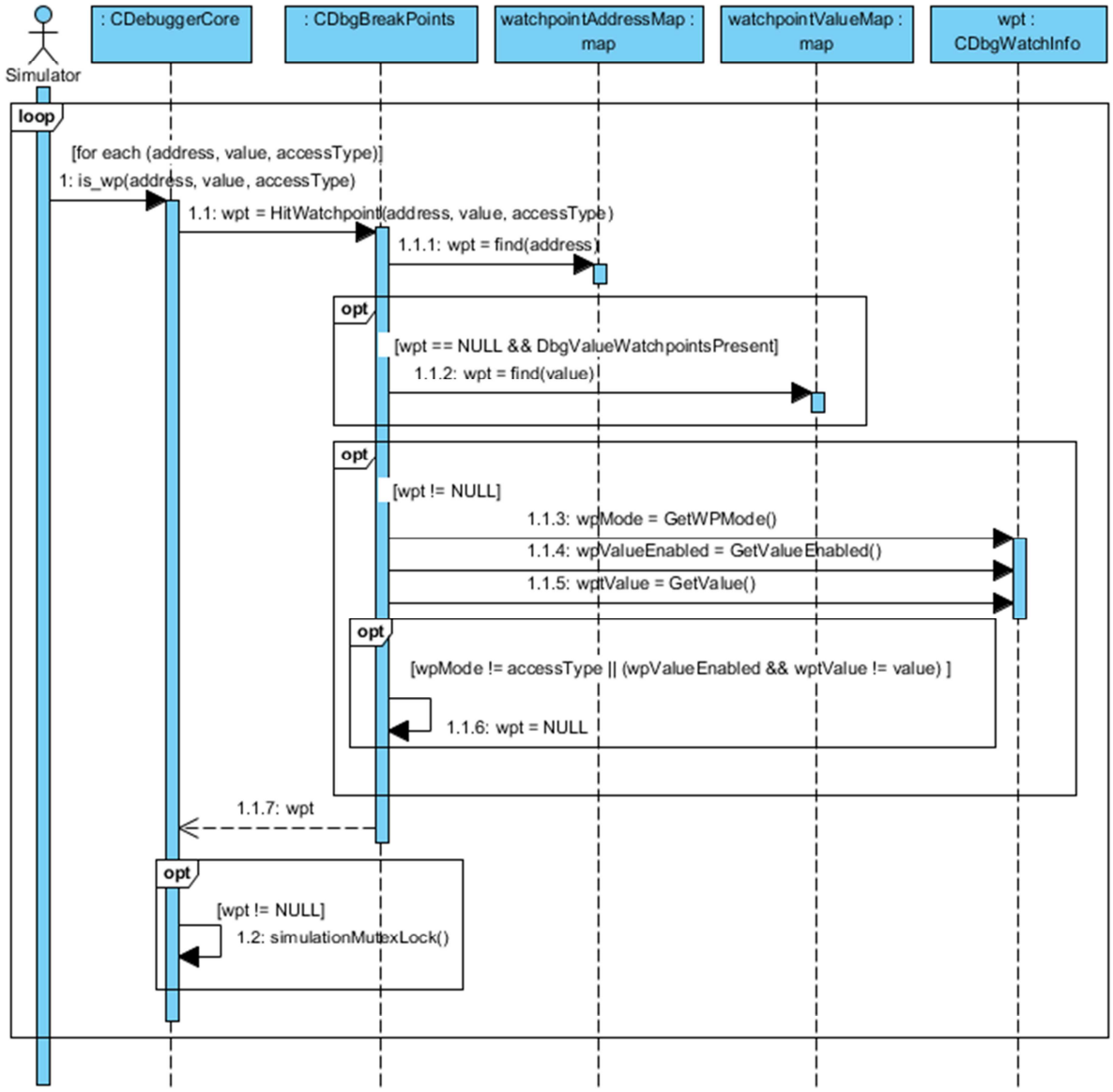
6.1.4.3 Úprava ladicí knihovny

Vzhledem k tomu, že vlastnosti instrukčních a datových breakpointů se mění v příkazové řádce pomocí stejných příkazů, bylo nutné upravit implementaci správce breakpointů tak, aby podporoval i datové breakpointy. Všechny společné vlastnosti instrukčních i datových breakpointů byly přesunuty do společné abstraktní třídy. Jedná se zejména o vlastnosti určující, zda je breakpoint povolený, počet jeho ignorovaných průchodů, nastavená podmínka a také číselný identifikátor, který se používá pro identifikaci breakpointu v příkazové řádce. Od této abstraktní třídy dědí třída reprezentující instrukční breakpoint a také třída reprezentující datový breakpoint. Zjednodušený diagram tříd zachycující strukturu reprezentace breakpointů je na obrázku 6.9.



Obrázek 6.9: Diagram tříd zachycující správce breakpointů v ladicí knihovně. Diagram byl zjednodušen, ve třídách byly vynechány některé vlastnosti a většina metod tak aby diagram popisoval základní strukturu.

Správce breakpointů obsahuje metodu, která na základě adresy, hodnoty a typu přístupu do paměti vrátí informace o datovém breakpointu, který způsobí zastavení provádění programu, přičemž se bere ohled na další vlastnosti breakpointu, jako nastavená podmínka nebo počet ignorovaných průchodů. Zjednodušený průběh komunikace mezi simulátorem a ladicí knihovnou při testování zásahu datového breakpointu je zachycen na obrázku 6.10.



Obrázek 6.10: sekvenční diagram zásahu datového breakpointu.

6.1.4.4 Propojení ladicí knihovny a simulátoru

Simulátor je propojený s ladicí knihovnou i po rozšíření stejně, jako bylo popsáno v kapitole 4.2. V základě byla jen rozšířena funkčnost pro podporu datových breakpointů. Simulátor volá ladicí knihovnu pomocí definovaného rozhraní před každou simulací následující instrukce programu. Nejprve je ladicí knihovna volána pro test, zda je simulátor v krokovacím módu. Pakliže není, je ladicí knihovna volána znovu pro test, zda se na adrese následující instrukce nachází instrukční breakpoint. Pokud ani tato podmínka není splněna a zároveň bylo v průběhu simulace předchozí instrukce přistoupeno do paměti, je ladicí knihovna volána znovu pro každý přístup do paměti (adresa, hodnota a typ přístupu). Simulace se případně pozastavuje přímo uvnitř ladicí knihovny pomocí mechanismu vzájemného vyloučení, kdy je znemožněno pokračování simulačního vlákna. Celý popis tohoto testování a volání ladicí knihovny je zkráceně naznačen v následujícím pseudokódu na obrázku 6.11.

```
sim_main()
{
  while(1) // hlavní smyčka simulace
  {
    ...

    if(dbg_is_step_mode(register_pc) == false)
    {
      if(dbg_is_bp(register_pc) == false)
      {
        if(DBG_WATCHPOINTS_IDX > 0) // došlo k přístupu do paměti
        {
          for (int i = 0; i < DBG_WATCHPOINTS_IDX; i++)
          {
            if (DBG_WATCHPOINTS[DBG_WATCHPOINTS_ADDRS[i]] ||
                DBG_VALUEWATCHPOINTSPRESENT)
                {
                  dbg_is_wp(DBG_WATCHPOINTS_ADDRS[i],
                   DBG_WATCHPOINTS_VALUES[i], DBG_WATCHPOINTS_TYPES[i]);
                }
          }
          DBG_WATCHPOINTS_IDX = 0;
        }
      }
    }

    ...

  }
}
```

Obrázek 6.11: pseudokód propojení simulátoru a ladicí knihovny. *register_pc* je proměnná, která simuluje registr čítače instrukcí a obsahuje tedy adresu následující instrukce.

6.2 Návrh chybějící funkcionality tříd 2 až 4 standardu Nexus

Pro třídu 2 je důležité zejména implementovat sledování toku programu. Pro implementaci této funkce bude nutné zejména modifikovat simulátor tak, aby v případě provedení instrukce skoku nebo při výjimce byla tato informace předána ladicímu nástroji a ten byl posléze schopen odeslat dané data pomocí zprávy přes střední vrstvu až do uživatelského rozhraní. Sledování vlastnictví nutné implementovat není, protože v rámci projektu není hotová podpora pro simulace operačních systémů.

Pro třídu 3 je požadována funkce pro zápis a čtení paměti bez zastavení procesoru. Tato funkce je lehce implementovatelná a nebude vyžadovat úpravy v kódu simulátoru, vzhledem k tomu, že komunikace s ladicím nástrojem probíhá v samostatném vlákně, které má přístup k paměti procesoru. Navíc lze použít příkazy *set* a *print*, pomocí kterých lze do paměti přistupovat i tehdy, pokud simulace běží. Pro podporu funkce sledování dat lze využít části funkcionality kódu, který bude implementován pro podporu datových breakpointů. Poté se jen po každé instrukci zkontroluje,

zda nebyl proveden přístup do sledovaného úseku paměti a případně se odešle příslušná zpráva. Další možnost je implementovat tuto funkci přímo v rámci kódu funkcí simulátoru pro přístup k paměti.

Čtvrtá třída požaduje jen funkci pro nahrazení paměti, ale vzhledem k tomu, že paměť je součástí simulátoru, není tato funkce pro simulátor potřebná.

6.3 Podpora pro ladění víceprocesorových systémů

Pro podporu ladění víceprocesorových systémů je nutné upravit funkcionalitu ladicího nástroje. Tyto požadavky nejsou přesně definovány ve standardu Nexus, proto bylo nutné implementovat všechny možné kombinace chování pří ladění.

6.3.1 Simulace víceprocesorových systémů

Pro popis ladění víceprocesorových systémů je nejprve nutné popsat princip samotné simulace víceprocesorových systémů. Ve víceprocesorové simulaci je pro každý procesor víceprocesorového systému vytvořen simulátor. Stejně jako v případě jednoprocesorové simulace je tento simulátor vygenerován z popisu architektury procesoru v modelu v jazyce ISAC. Každý simulátor je samostatně spustitelný program a každý může být spuštěn na jiném počítači. Samotné spuštění jednotlivých simulátorů má na starosti střední vrstva, která z konfigurace specifikované XML souborem provede spuštění, inicializaci a navázání komunikace se simulátory.

Komunikace mezi střední vrstvou a jednotlivými simulátory probíhá ve dvou kanálech na bázi protokolu TCP/IP. První kanál je určen pro synchronní komunikaci a je otevřen na portu specifikovaném v XML konfiguraci pro instalaci simulátorů. Druhý kanál je určen pro asynchronní zprávy ze simulátorů a je otevřen na portu o jedno vyšším než port pro synchronní komunikaci. Toto je nutné brát v potaz při konfiguraci více simulátorů na jednom počítači a přiřadit v konfiguraci pro druhý simulátor port minimálně o dvě vyšší než je port pro první simulátor. V opačném případě instalace simulátorů selže.

Navázání komunikace mezi simulátory proběhne až poté, co je z příkazové řádky odeslán příkaz pro spuštění simulace (*DBL_DEBUG_ALL* nebo *DBL_START_ALL*). Po obdržení tohoto příkazu střední vrstva rozešle příkaz pro spuštění všem jednotlivým simulátorům a v rámci parametrů tohoto simulátoru je předán i seznam všech nainstalovaných simulátorů. Tento seznam obsahuje seznam identifikátoru, hostitele a portu pro každý nainstalovaný simulátor. Pomocí těchto parametrů pak může každý simulátor navázat spojení s ostatními simulátory.

6.3.1.1 Asynchronní simulace

V konfiguračním souboru pro víceprocesorovou simulaci lze pro každý procesor specifikovat frekvenci. Pokud je tato frekvence nastavena na nulovou hodnotu, jedná se o asynchronní simulaci.

V asynchronní simulaci procesory běží relativně nezávisle. Znamená to, že jeden procesor nečeká na provedení instrukcí v druhém procesoru. Vynechání synchronizace umožňuje relativně rychlý běh simulace, zvlášť když jsou simulátory nainstalované na různých počítačích a simulace je takto rozdistribuována. Na druhou stranu ale tato simulace nemodeluje přesně chování víceprocesorového systému, kde je nutná synchronizace přístupu na sběrnici. Naopak pro

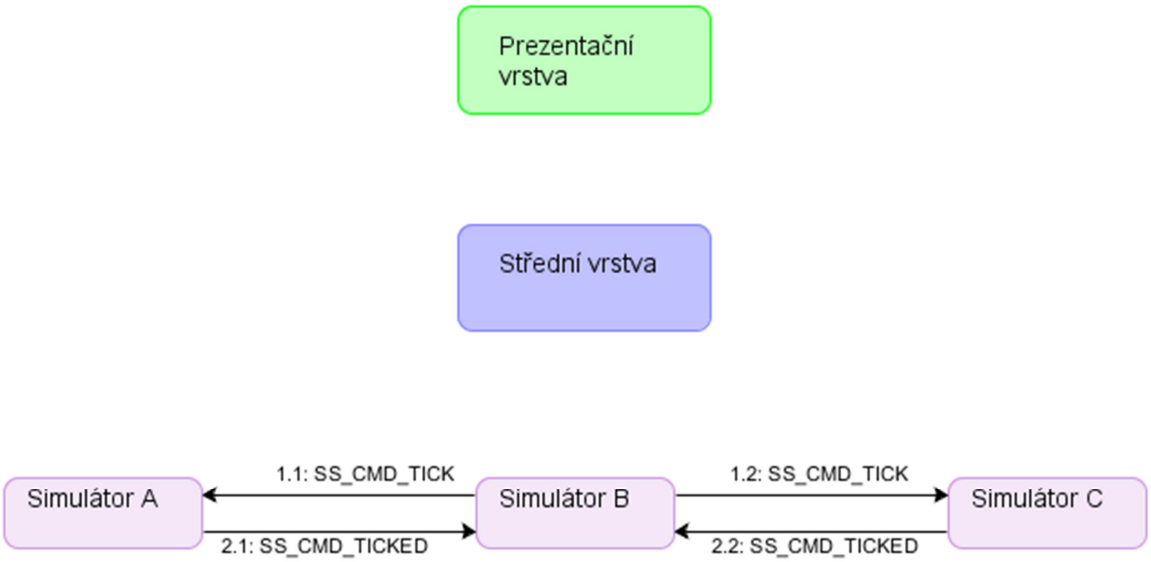
víceprocesorové systémy založené na architektuře Síť na čipu je asynchronní simulace vhodná, protože v těchto systémech také běží jádra jednotlivých procesorů relativně nezávisle.

6.3.1.2 Synchronní simulace

Druhou variantou je synchronní simulace. V této variantě je nutné kromě specifikace frekvencí jednotlivých procesorů v konfiguračním souboru také vygenerovat simulátor s podporou synchronizace.

Synchronní simulace je oproti asynchronní simulaci přesnější, na druhou stranu ale časově mnohem náročnější, díky nutnosti synchronizace procesorů. Samotná synchronizace funguje na principu, kdy jeden simulátor je tzv. hlavní simulátor, který generuje hodinový signál pro ostatní simulátory. Ty tedy pro provedení následující instrukce vždy čekají na signál od hlavního simulátoru a hlavní simulátor vždy čeká na dokončení předchozí instrukce ostatních simulátorů, než vygeneruje nový hodinový impuls.

Samotná synchronizace je implementována pomocí mechanismů sdílené paměti, ale tento mechanismus je dostupný jen pokud se simulace spouští na platformě Linux a všechny simulátory jsou nainstalovány na jednom počítači. V opačném případě se pro synchronizaci použije komunikace přes síť pomocí protokolu TCP/IP.



Obrázek 6.12: Schéma komunikace při běhu synchronní simulace v případě, kdy není k synchronizaci použita sdílená paměť. Komunikace pro synchronizaci probíhá pouze mezi simulátory. Simulátor B je hlavní simulátor, který generuje hodinový signál.

6.3.2 Ladění víceprocesorových systémů

6.3.2.1 Výběr simulátoru v příkazové řádce

První problémem při ladění víceprocesorových systémů je rozlišení jednotlivých procesorů v rámci simulace víceprocesorového systému. Každý simulátor má přidělen identifikátor v konfiguračním XML souboru. Tento identifikátor pak slouží i v příkazové řádce k identifikaci simulátoru.

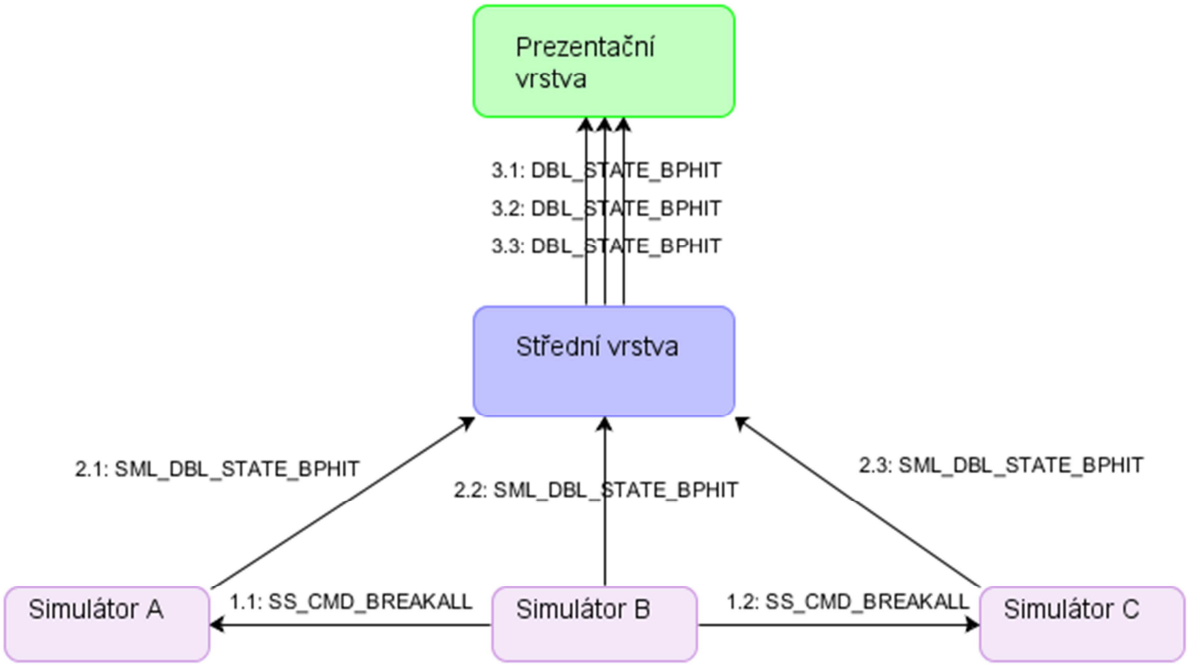
V příkazové řádce byl implementován mechanismus výběru simulátoru. To znamená, že vždy je vybraný jeden nainstalovaný simulátor, kterému střední vrstva přeposílá všechny příkazy – nastavení breakpointu, krokování apod. Identifikátor simulátoru je vždy vložen do zprávy pro střední

vrstvu jako parametr. Střední vrstva pak pomocí tohoto identifikátoru najde v tabulce spojení s konkrétním simulátorem a příkaz mu odešle. Ovšem existují i příkazy, které se zasílají všem simulátorům. Jedním z nich je například příkaz pro spuštění simulace. V tomto případě pak identifikátor simulátoru ve zprávě chybí.

Samotný výběr simulátoru se provede pomocí příkazu *simulator*, který má jediný parametr identifikátor simulátoru. Pokud se tento parametr vynechá, příkaz vypíše aktuálně vybraný simulátor. Kromě tohoto výběru simulátoru pomocí příkazu, byla navíc implementována možnost použít jméno simulátoru jako předponu příkazu. Použití takového příkazu se prakticky rovná zkratce, která provede přepnutí simulátoru, spuštění příkazu a přepnutí zpět na původní simulátor. Použití tohoto příkazu je vhodné, když je potřeba zadat větší množství příkazů, z nichž každý je určen pro jiný simulátor.

6.3.2.2 Breakpointy ve víceprocesorové simulaci

Další problém, který je nutno řešit pro víceprocesorové systémy, je co se má stát, pokud jeden z procesorů narazí na breakpoint. Možnosti jsou prakticky jen dvě: zastaví se jen daný procesor, na kterém byl nastaven breakpoint, nebo se zastaví všechny procesory. Proto bylo nutné přidat vlastnost breakpointu, která určuje, zda se pro aktivaci daného breakpointu mají zastavit i ostatní simulátory. Tato vlastnost je vždy ve výchozím stavu zapnuta. Její stav lze měnit pomocí příkazů *enablebreakall* a *disablebreakall*, které pro daný breakpoint tuto vlastnost zapnou nebo vypnou.



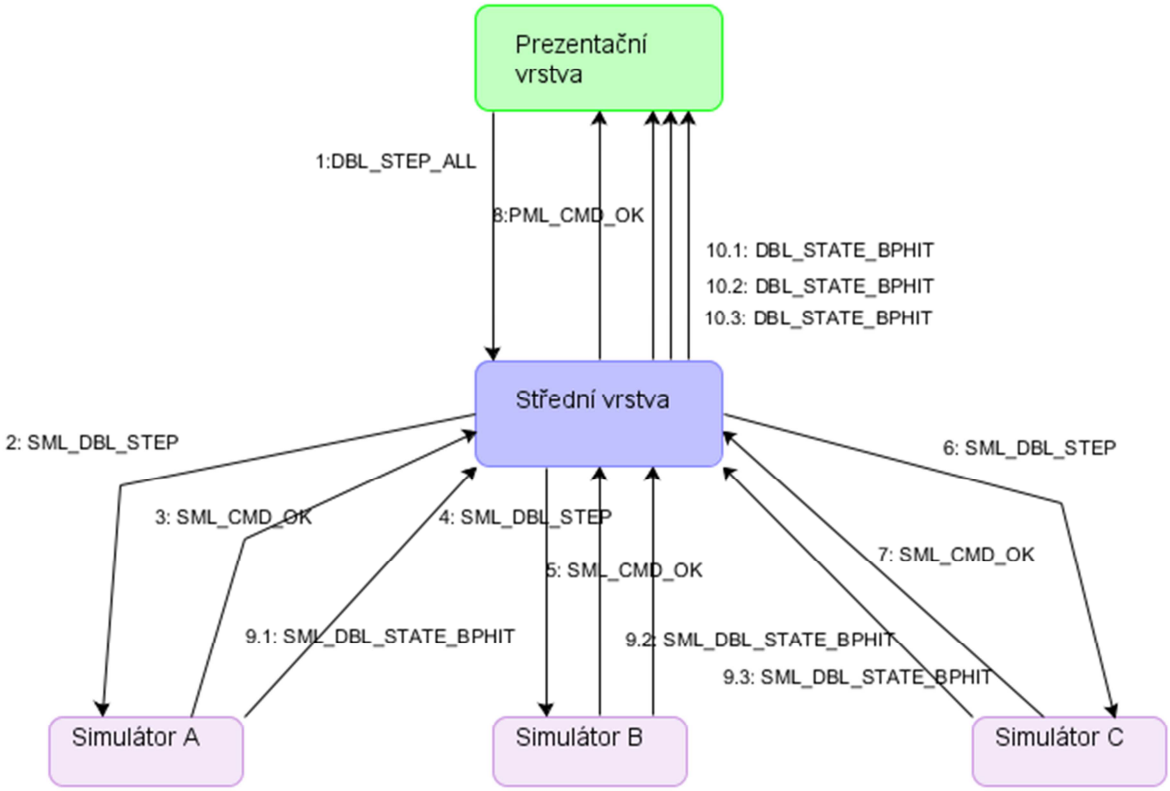
Obrázek 6.13: Schéma komunikace mezi jednotlivými vrstvami a simulátory v asynchronní simulaci, pokud procesor simulovaný simulátorem B zastaví na breakpointu a zároveň je u tohoto breakpointu nastavena vlastnost zastavení všech ostatních procesorů

Rozeslání zprávy, která zastaví ostatní procesory, je implementováno v rámci funkce ladicí knihovny, která testuje, zda se na dané adrese nachází breakpoint. Pokud je breakpoint nalezen, tedy aktivován, je všem ostatním simulátorům rozeslán příkaz *SS_CMD_BREAKALL*, který nastaví simulátor do krokovacího módu.

6.3.2.3 Krokování víceprocesorové simulace

Po zastavení simulace na breakpointu většinou uživatel chce program krokovat. Ale zde vyvstává otázka, zda krokovat jenom ten procesor, který se zastavil breakpointem, nebo všechny zastavené procesory. Proto byl do příkazové řádky doplněn nový příkaz *stepall*, který krokuje všechny procesory. Tento příkaz funguje stejně jako příkaz *step*, jen střední vrstva rozesílá zprávu všem simulátorům. Stávající příkaz *step* pro krokování krokuje jen aktuálně vybraný procesor.

Se zastavenou simulací také souvisí problém s pokračováním simulace. Uživatel musí mít možnost spustit jen některé procesory, nebo spustit pokračování simulace všech procesorů. Pro spuštění jen jednoho procesoru může použít stávající příkaz *continue*, který způsobí pokračovaní simulace aktuálně vybraného simulátoru. Pro pokračování simulace všech procesorů byl přidán příkaz *continueall*.



Obrázek 6.14: Schéma komunikace mezi jednotlivými vrstvami a simulátory v asynchronní simulaci, pokud jsou všechny simulátory v krokovacím módu a uživatel v prezentační vrstvě zadá příkaz pro provedení jednoho kroku simulace na všech procesorech.

6.3.2.4 Synchronní víceprocesorová simulace

Synchronní víceprocesorová simulace má své specifika oproti asynchronní víceprocesorové simulaci. Prvním specifikem je chování breakpointů. Pokud jeden procesor narazí na breakpoint, musí se zastavit celá simulace, tedy všechny procesory. Je to z toho důvodu, že pokud by se zastavil jenom jeden, stejně se zastaví všechny procesory, protože budou čekat na dokončení provádění instrukce zastaveného procesoru a díky synchronizaci nezačnou provádět další následující instrukci. Proto se po zasažení breakpointu vždy rozešle zpráva všem ostatním simulátorům a všechny simulátory se přepnou do krokovacího módu. Příkazy *enablebreakall* a *disablebreakall* tedy v synchronní simulaci postrádají svůj význam a nelze je použít.

S přepnutím všech simulátorů do krokovacího módu souvisí i samotné krokování. Nemá smysl krokovat jen jeden procesor, vždy se musí příkaz pro krokování zaslat všem procesorům. Z toho důvodu není možné v synchronní víceprocesorové simulaci použít příkaz *step*, ale je možné použít jen příkaz *stepall*. Ovšem zde nastává jeden malý problém, pokud jeden procesor má například poloviční frekvenci než druhý. Poté po obdržení příkazu pro krokování rychlejší procesor provede krok simulace, pomalejší také provede svůj krok simulace, ale díky synchronizaci bude čekat na provedení následujícího kroku simulace až poté, co rychlejší procesor provede další krok simulace. Proto následující použití příkazu *stepall* provede prakticky krokování jen toho rychlejšího procesoru.

S krokováním také souvisí problém pokračování simulace. Ve víceprocesorové synchronní simulaci opět nemá význam spustit jen jeden procesor, ale musí se spustit pokračování všech simulátorů na jednou. Proto nelze použít příkaz *continue*, ale jen příkaz *continueall*.

6.3.2.5 Použití výrazů ve víceprocesorové simulaci

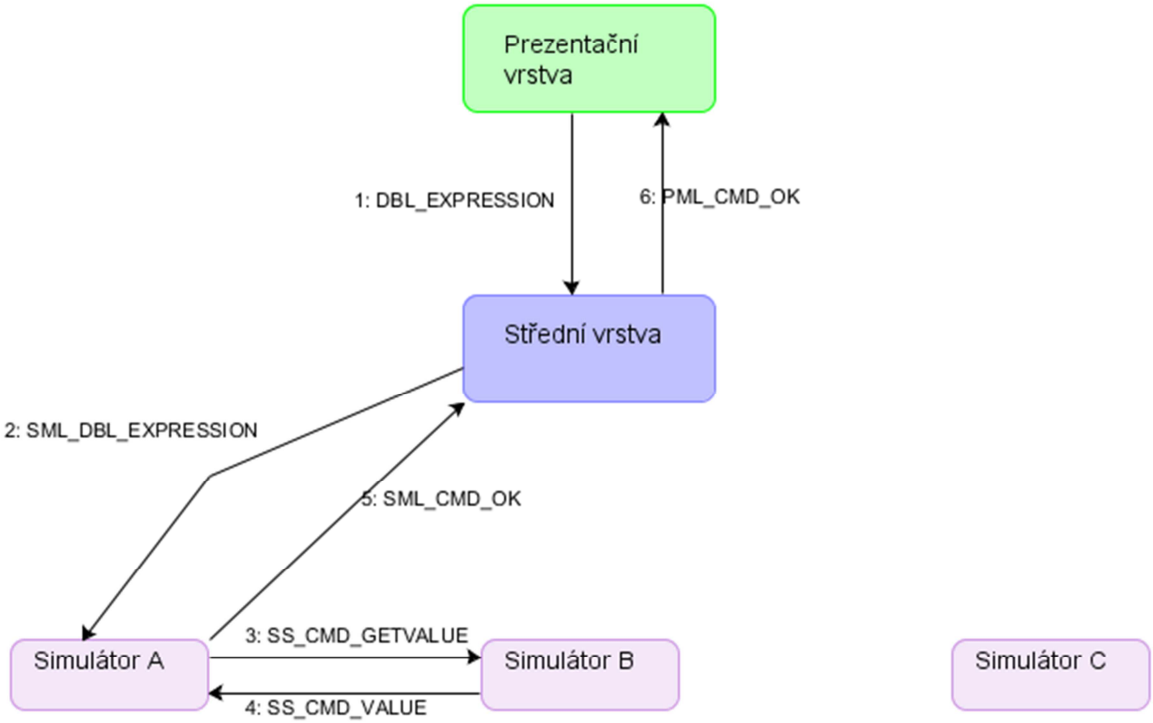
Modul pro zpracování výrazů zpracovává výrazy zadané v textové podobě v příkazové řádce pomocí příkazů *set* a *print*, a také zpracovává výrazy, které jsou nastavené jako podmínka instrukčního nebo datového breakpointu.

Vzhledem k tomu, že pro vyhodnocení výrazů je potřeba přistupovat přímo ke zdrojům simulátoru, je tento modul implementován v rámci ladicí knihovny. Textová podoba výrazů je zasílána simulátoru a ladicí knihovně ke zpracování pomocí příkazu DBL_EXPRESSION případně pomocí příkazu DBL_BREAKPOINT, pokud je výraz nastaven jako podmínka breakpointu.

Před vyhodnocením výrazu je nutné výraz přeložit. K tomu slouží překladač výrazu, který výraz z textové podoby převede pomocí lexikální a syntaktické analýzy na výraz ve formě stromu. Nad tímto stromem pak lze volat metodu pro vyhodnocení jednotlivých uzlů. Jednotlivé hodnoty jsou v uzlech ukládány. To znamená, že v případě nutnosti znovu vyhodnocení hodnoty uzlu je třeba tuto hodnotu resetovat. Resetování hodnoty se volá rekurzivně, takže jsou resetovány také hodnoty všech podřízených uzlů.

Jednotlivé uzly jsou různých typů, přičemž jedním z těchto typů je i typ uzlu reprezentující zdroj procesoru. Tento zdroj se může nacházet obecně v libovolném procesoru. K identifikaci simulátoru, ke kterému patří daný zdroj, byl zvolen operátor tečka. Například zobrazení hodnoty registru *pc* v procesoru *mips_basic* se provede pomocí následujícího příkazu: *print ${mips_basic.pc}*. Během syntaktické analýzy tedy syntaktický analyzátor přiřazuje všem uzlům typu zdroj jméno simulátoru. Pokud takový uzel simulátor nemá nastaven, znamená to, že se jedná o lokální zdroj.

Vyhodnocování výrazů pak probíhá tak, že se rekurzivně vyhodnocují uzly od kořene stromu. Pokud se narazí na uzel typu zdroj, který se nachází v jiném simulátoru, je tomuto simulátoru zaslán příkaz *SS_CMD_GETVALUE*. Cílový simulátor tento příkaz zpracuje, vyhodnotí hodnotu požadovaného zdroje a zašle ji zpět zdrojovému simulátoru. Ten tuto hodnotu použije dál při zpracování příkazu.



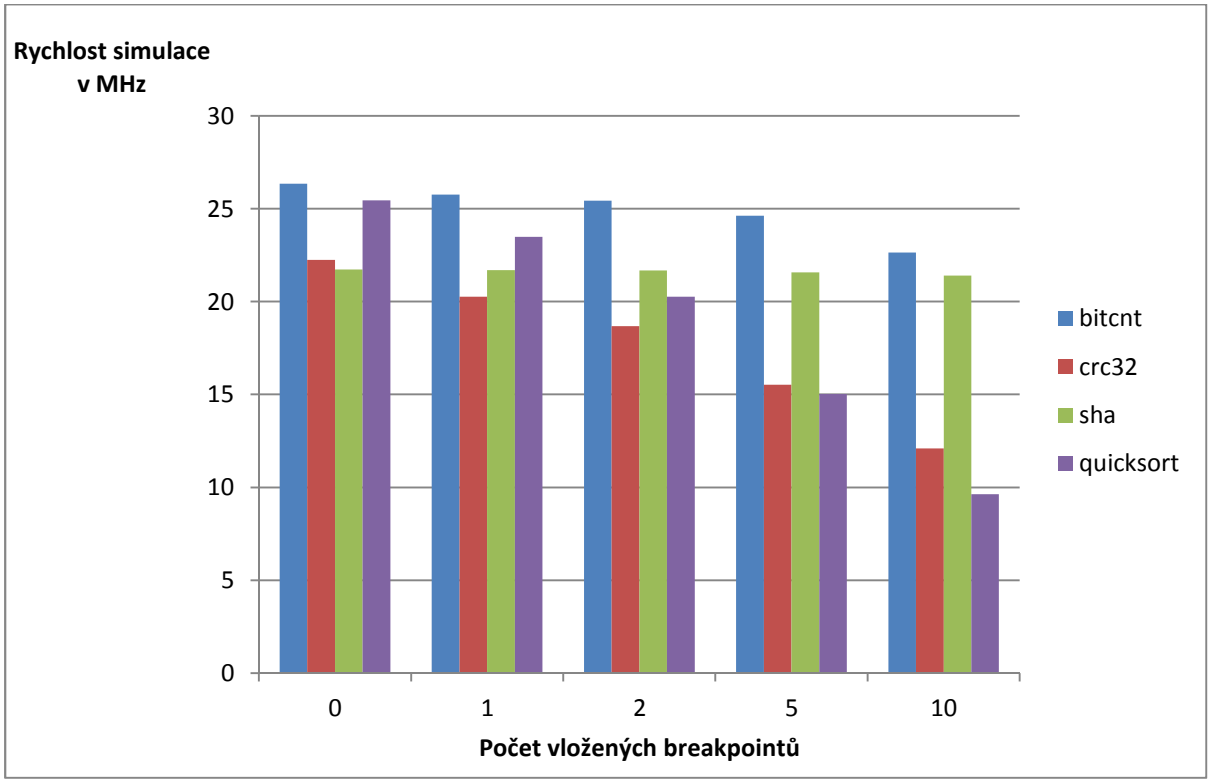
Obrázek 6.15: Schéma komunikace mezi jednotlivými vrstvami a simulátory v asynchronní simulaci, pokud uživatel požaduje vyhodnocení výrazu pomocí příkazu *print* nebo *set*, který obsahuje i hodnotu zdroje, který se nachází v procesoru simulovaném v simulátoru B

7 Testování a vyhodnocení ladicího nástroje

7.1 Vliv breakpointů na rychlost simulace

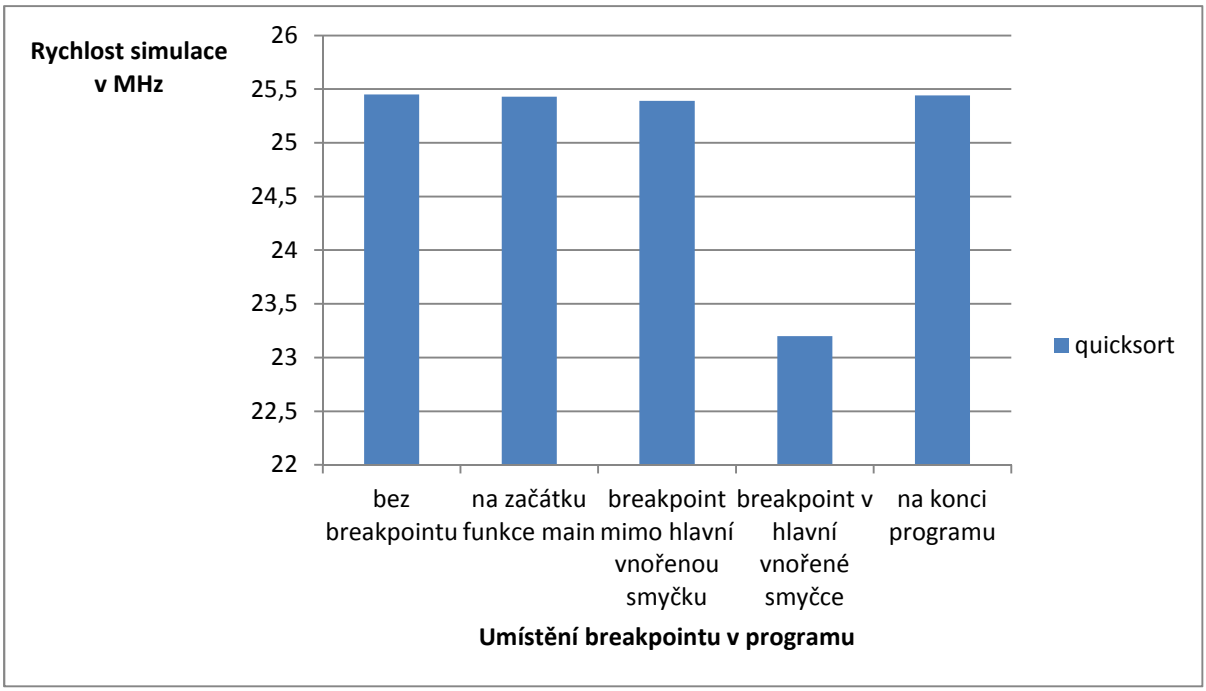
Jednou z mála metrik, na základě kterých se dá posuzovat výkon ladicího nástroje je v případě projektu Lissom vliv ladicí knihovny na rychlost simulace. Jedním z těchto vlivů je počet breakpointů v simulovaném programu. Breakpointy mají na rychlost simulace vliv z toho důvodu, že před provedením následující instrukce je volána ladicí knihovna, ve které je testováno, zda se na dané adrese nachází breakpoint či ne. Zpomalení tedy nastává hlavně z důvodů nalezení daného breakpointu podle adresy a pak vyhodnocení, zda se na breakpointu má simulace zastavit či ne. Toto vyhodnocení se skládá z vyhodnocení parametrů breakpointu, jako je počet ignorování, pak vyhodnocení stavu breakpointu (zda je povolený či ne) a také vyhodnocení podmínky breakpointu.

Pro testování jsem vybral procesor MIPS[19], na kterém jsem postupně testoval relativně jednoduché programy bitcnt, crc32, sha a quicksort. Do těchto programů pak byly před simulací vloženy postupně breakpointy na lokalitu, kde dochází k největšímu množství zásahu breakpointů. U všech těchto breakpointů byla nastavena podmínka na ignorování breakpointu na relativně vysokou hodnotu, tak aby nedocházelo k zastavení simulace. Vliv těchto breakpointů na rychlost simulace je uveden v grafu na obrázku 7.1.



Obrázek 7.1: graf znázorňující závislost rychlosti simulace na programu a nastaveném počtu breakpointů

Jak je patrné z grafu na obrázku 7.1, na rychlost simulace má kromě počtu breakpointů vliv také charakter programu, to znamená, zda jak často dochází k zásahu breakpointu. Z toho důvodu jsem provedl ještě jeden test nad programem quicksort, kde jsem jeden breakpoint umísťoval na různé místa v programu. Zvolil jsem umístění začátek a konec programu, kde dojde prakticky k jedinému zásahu a pak na dvě různé místa v hlavní části algoritmu, přičemž jedno bylo umístěno ve vnořeném cyklu. Výsledky tohoto testu jsou znázorněny v grafu na Obrázku 7.2 a pro lepší přehlednost také v tabulce 7.1



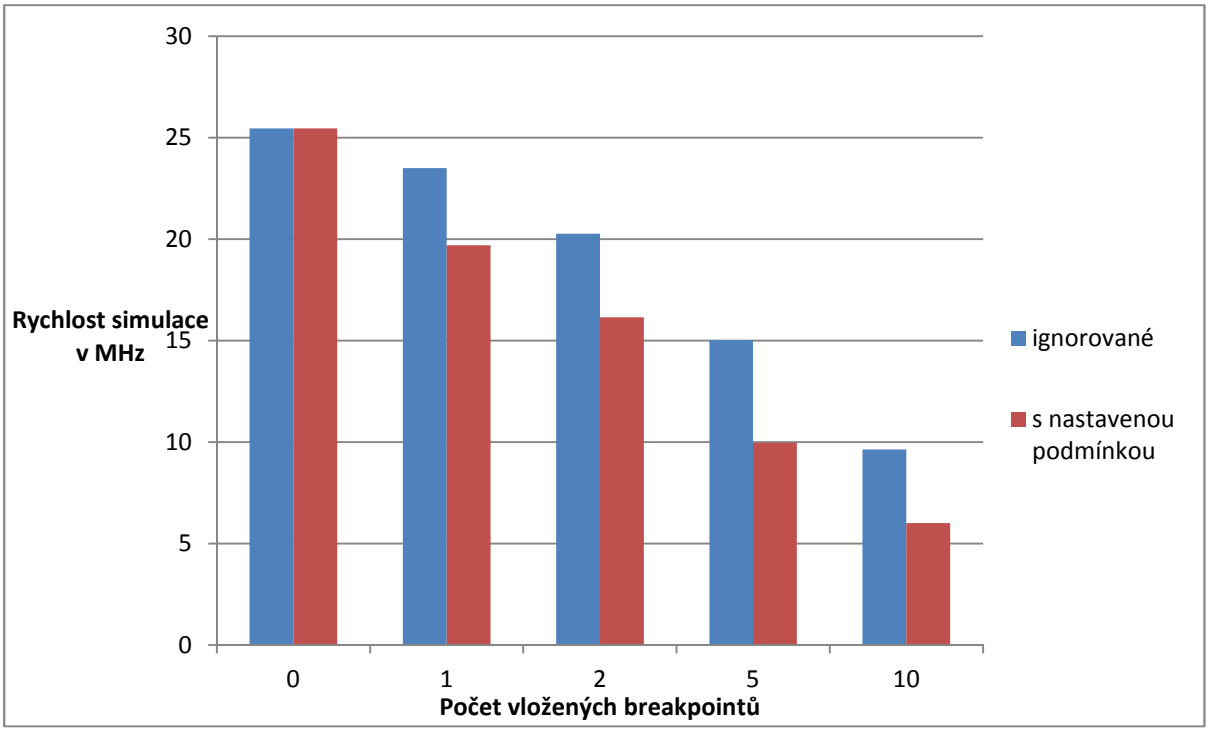
Obrázek 7.2: graf znázorňující závislost rychlosti simulace programu quicksort na umístění breakpointu

bez breakpointu	na začátku funkce main	breakpoint mimo hlavní vnořenou smyčku	breakpoint v hlavní vnořené smyčce	na konci programu
25,45	25,43	25,39	23,2	25,44

Tabulka 1: rychlosti simulace v MHz v závislosti na umístění breakpointu v programu quicksort

Z naměřených údajů vyplývá, že jediný breakpoint má vliv na rychlost simulace, jen pokud dojde k velkému počtu jeho zásahů. Jediný zásah breakpointu nemá praktický vliv na rychlost simulace.

Na rychlost simulace také má vliv podmínka, která může být nastavena pro breakpoint. Proto jsem pro další test zvolil porovnání rychlostí simulace programu quicksort s různým počtem breakpointů, kdy v jednom případě nastavil pouhou ignoraci breakpointů a ve druhém jsem vždy pro každý breakpoint nastavil podmínku, která se vždy po zásahu breakpointu vyhodnocovala. Podmínka byla zvolena tak, aby nikdy nedošlo k zastavení simulace. Výsledky tohoto porovnání na programu quicksort a vliv vyhodnocování podmínek na rychlost simulace je znázorněn v grafu na obrázku 7.3

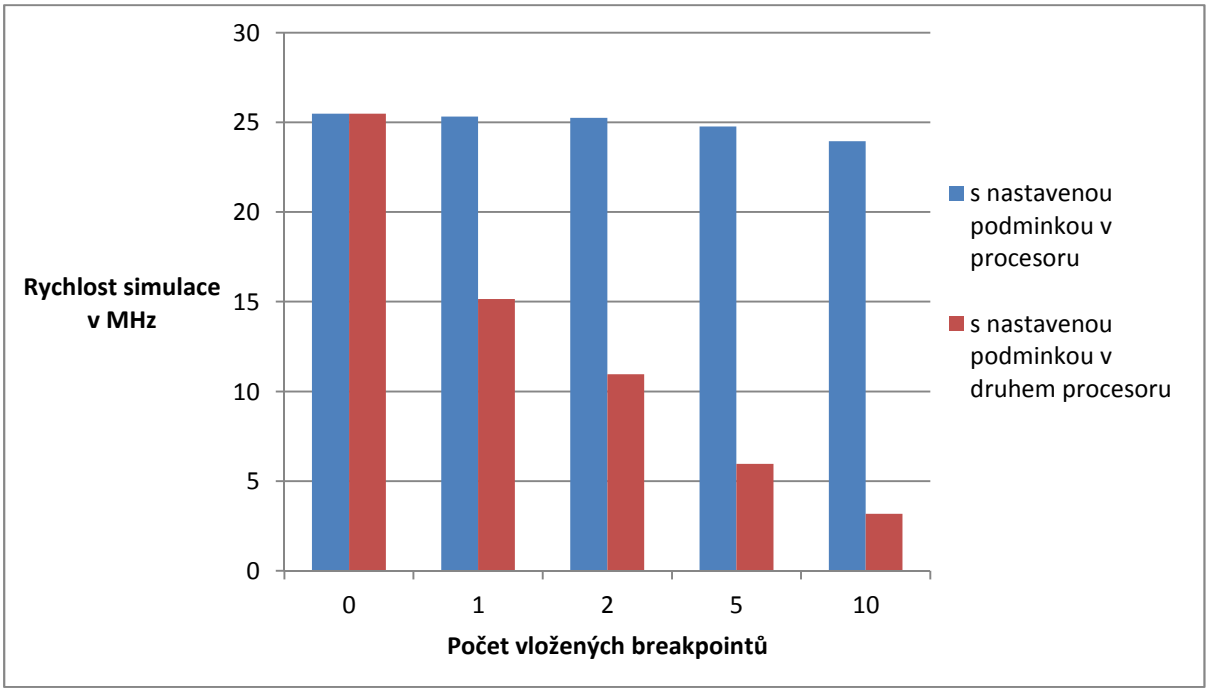


Obrázek 7.3: Porovnání zpomalení rychlosti simulace pokud mají breakpointy nastavenou podmínku

7.1.1 Breakpointy ve víceprocesorové simulaci

Ve víceprocesorové simulaci je rychlost simulace také ovlivněna ladicí knihovnou. Pokud simulátory mezi sebou nekomunikují, je rychlost simulace stejná jako v případě jednoprocesorové simulace za předpokladu, že každý program simulátoru běží na vlastním procesoru – to je splněno buď v případě použití víceprocesorového počítače, nebo pomocí distribuované simulace.

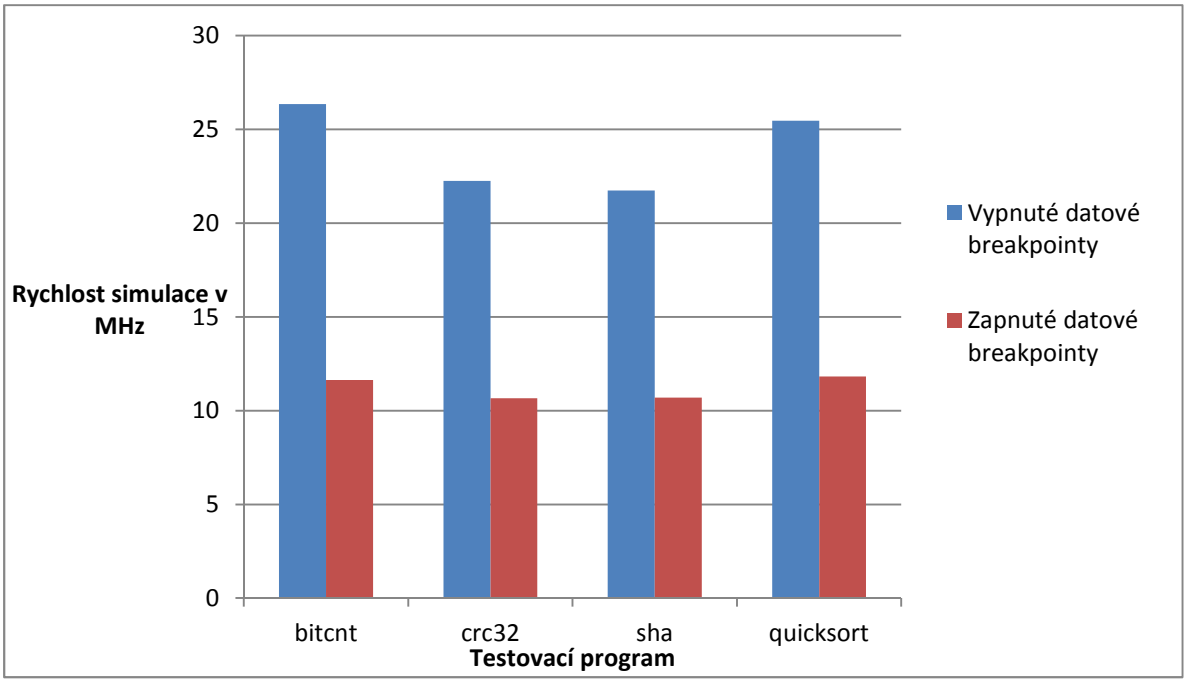
V případě, kdy v rámci simulace musí simulátory mezi sebou komunikovat, je zpomalení velice znatelné. Jako příklad uvádím test, kdy jsem použil jednoduchý víceprocesorový systém skládající se ze dvou procesorů MIPS. Na každém procesoru běžel stejný program quicksort a do jednoho z nich jsem umístil postupně několik breakpointů. Simulace běžela asynchronně. Z důvodu znatelného snížení rychlosti simulace jsem breakpointy umístil do místa programu, kde nedocházelo k častému zásahu breakpointu. V obou případech jsem nastavil všem breakpointům podmínku. V prvním případě se pro vyhodnocení podmínky nemusel kontaktovat žádný jiný simulátor, ale v druhém případě byl v podmínce použit zdroj z druhého procesoru, takže pro vyhodnocení podmínky musel simulátor zaslat zprávu druhému simulátoru. Z důvodu, že simulace byla asynchronní, tak simulace procesoru bez breakpointu běžela prakticky stejně rychle jako simulace jednoprocesorového systému bez nastavených breakpointů. Porovnání rychlosti simulace druhého procesoru je znázorněna na grafu na obrázku 7.4



Obrázek 7.4: zpomalení simulace ve víceprocesorové simulaci, pokud breakpointy obsahují podmínku, pro jejíž vyhodnocení je třeba kontaktovat jiný simulátor

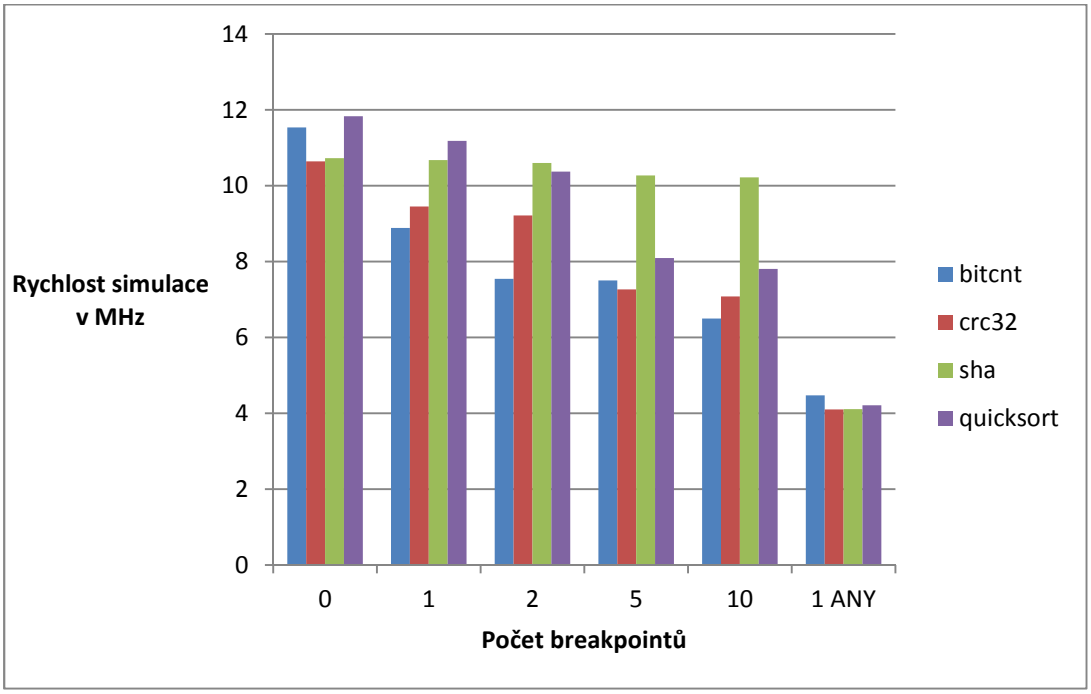
7.2 Vliv datových breakpointů na rychlost simulace

Datové breakpointy mají na rychlost simulace větší vliv než instrukční breakpointy. Je to z toho důvodu, že simulátor musí zaznamenávat všechny přístupy do paměti. Proto byl modifikován generátor simulátoru tak, aby tento kód na sledování přístupu do paměti byl generován jen tehdy, pokud si to uživatel přeje. Tento kód zaznamenává přístupy i tehdy, pokud není žádný datový breakpoint vložen. Proto dojde ke znatelné ztrátě na rychlosti simulace, pokud je podpora pro datové breakpointy požadována při generování simulátoru. Toto zpomalení znázorňuje graf na obrázku 7.5, kde byl simulátor procesoru MIPS testován s různými programy.



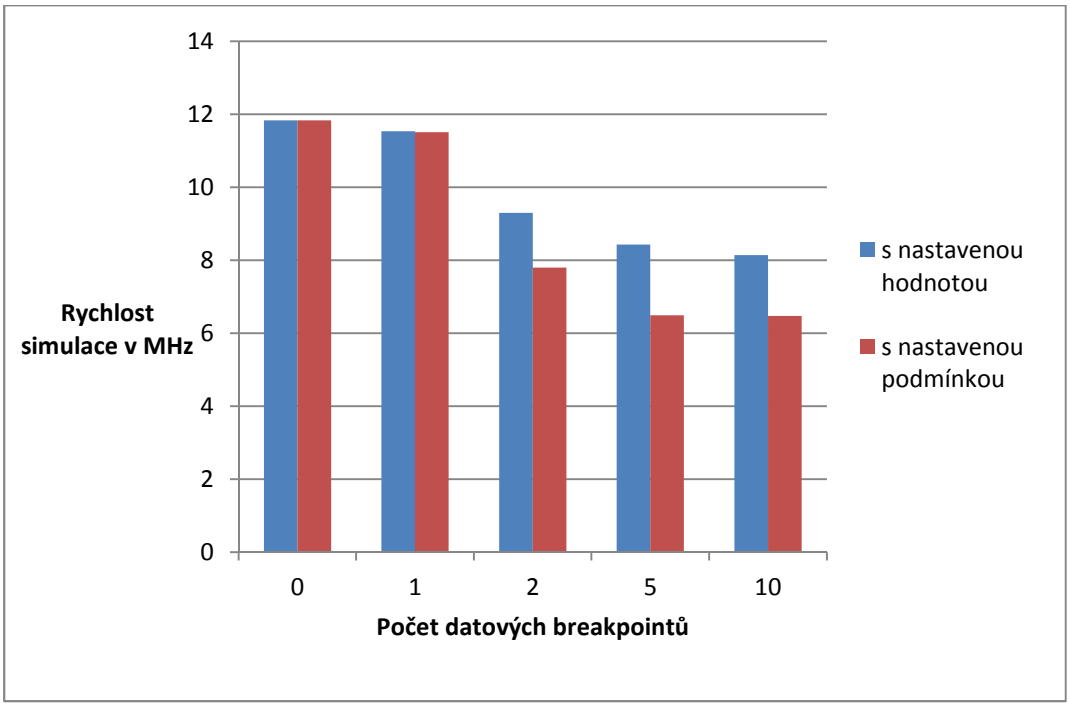
Obrázek 7.5: Graf zobrazující rychlost simulace bez breakpointů – s vypnutou podporou datových breakpointů a se zapnutou podporou datových breakpointů

Kromě samotného sledování přístupu do paměti je simulace zpomalována také i vyhodnocováním těchto přístupů do paměti. Toto vyhodnocování je tím pomalejší, čím více je vloženo datových breakpointů. Samozřejmě také záleží i na charakteru aplikace, to znamená, jak často přistupuje do paměti. Vliv počtu vložených datových breakpointů na rychlost simulace znázorňuje následující graf na obrázku 7.6. *1 ANY* znázorňuje jeden vložený datový breakpoint, který nemá definovanou adresu, na kterou má hlídat přístupy. Je tudíž vyhodnocován pro každý přístup.



Obrázek 7.6: graf znázorňující závislost rychlosti simulace na počtu vložených datových breakpointů.

Datové breakpointy umožňují i nastavit sledovanou hodnotu ke sledované adrese a zastavit program jen v případě, pokud se na danou adresu zapisovala nebo četla definovaná hodnota. Pro další test jsem tedy zvolil porovnání simulace s různým počtem datových breakpointů, kdy v jednom případě měly všechny tyto breakpointy nastavenou sledovanou hodnotu a v druhém případě měly místo této hodnoty nastavenou standardní podmínku. Srovnání je znázorněno na grafu na obrázku 7.7. V tomto grafu lze zároveň vidět, jaký má vliv počet zásahů datového breakpointu na rychlost simulace. V případě, kdy byly vloženy dva breakpointy, došlo ke znatelnému propadu rychlosti, přičemž první z těchto dvou breakpointů byl vložen na stejnou adresu jako v případě jediného breakpointu.



Obrázek 7.7: vliv datových breakpointů s podmínkou na rychlost simulace

7.3 Metoda testování

Všechny testy byly provedeny na stejném počítači. Jednalo se o stanici, na které běžel 64bitový operační systém Linux. Tato stanice měla procesor Intel Core 2 Quad běžící na frekvenci 2.8 GHz a 4GB operační paměti. Pro generování programů simulátorů byl použit kompilátor GCC se zapnutými optimalizacemi (-O3).

Výsledky zobrazené v předchozích tabulkách a grafech reprezentují průměrné výsledky, které byly získány z hodnot získaných z pěti spuštění programu simulace. Průměrná odchylka rychlostí simulace při jednotlivých spuštěních simulace od průměru byla v řádu setin Mhz.

8 Návrh rozšíření ladicího nástroje

Předchozí kapitoly se zaobíraly problematikou ladicího nástroje v projektu Lissom, který umožňuje ladění programů běžících v simulátorech. Ovšem v reálném produkčním prostředí je vhodné, aby bylo možné ladit i programy, které běží v prototypech procesorů v reálném hardwaru. V této kapitole je popsán návrh rozšíření ladicího nástroje tak, aby umožňoval takovéto ladění.

8.1 Uživatelské rozhraní

Z pohledu uživatele ladicího nástroje musí být uživatelské rozhraní transparentní. To znamená, že pro oba případy ladění programu (simulace i reálný hardware) budou použity naprosto stejné příkazy se stejnou syntaxí parametrů. Je ovšem možné, že některé příkazy nebude možno použít v případě ladění v hardwaru. V takovémto případě uživatelské rozhraní jen zobrazí uživateli zprávu, že daný příkaz není pro tento způsob ladění podporován. Mezi takovéto příkazy bude patřit zcela jistě například příkaz pro nastavení podmínky pro breakpoint, jelikož reálný procesor nebude schopen tyto podmínky vyhodnocovat.

8.2 Knihovna pro komunikaci s hardware

S uživatelským rozhraním souvisí i problém, do které vrstvy třívrstvé architektury umístit knihovnu, v rámci které bude implementována komunikace s hardwarem pomocí Nexus protokolu přes AUX port laděného zařízení. Nabízí se prakticky dvě možnosti. První možnost je umístit tuto knihovnu přímo do vrstvy uživatelského rozhraní, které by pak mělo dva módy funkčnosti (simulační a ladění v hw) a podle nastaveného módu by buď volalo zpracování příkazu pomocí Nexus knihovny nebo příkaz přeposlalo střední vrstvě v případě ladění v simulátoru. Tato možnost má ale zásadní nevýhodu, protože na vrstvě uživatelského rozhraní jsou implementovány dvě aplikace – příkazová řádka a zásuvný modul pro platformu Eclipse. Každá z těchto dvou aplikací je ovšem implementována pomocí jiné technologie (nativní kód C++ pro příkazovou řádku, Java pro modul pro Eclipse), což by znamenalo nutnou dvojí implementaci této Nexus knihovny. Částečným řešením by bylo použití knihovny, která obaluje a zpřístupňuje C++ kód v Javě, ale toto řešení není ideální. Jako ideálnější a jednodušší řešení se proto jeví umístění přístupu k hardwaru do střední vrstvy, takže nebude potřeba měnit prezentační vrstvu, jen bude potřeba změnit implementaci middleware.

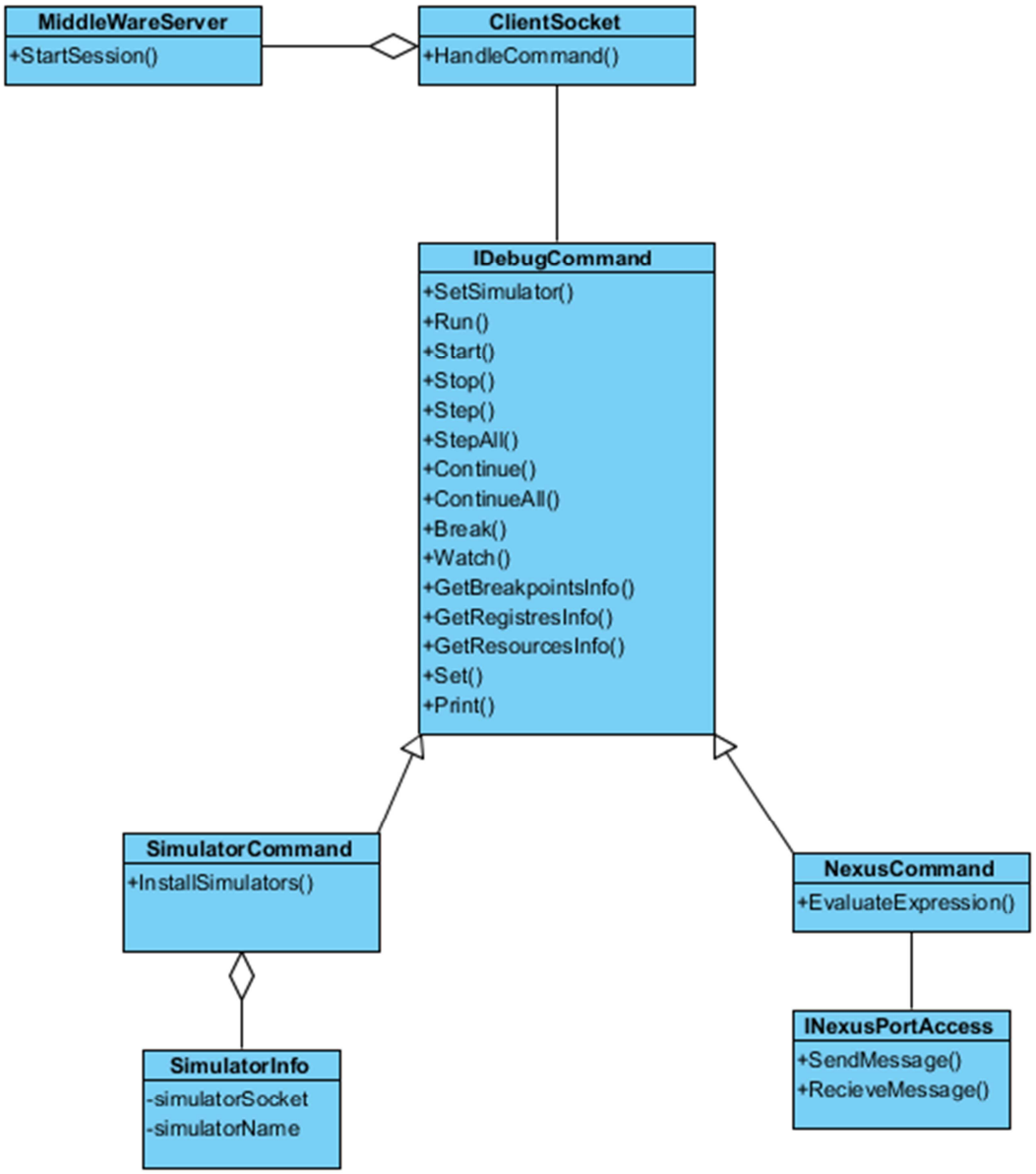
8.3 Návrh implementace rozšíření střední vrstvy

Současná implementace střední vrstvy zpracovává příkazy z prezentační vrstvy. Tyto příkazy se dají rozdělit do dvou kategorií. Jednak jsou to příkazy určené přímo střední vrstvě (např. instalace simulátorů nebo spuštění generátoru nástrojů) a pak příkazy určené k přeposlání simulátorům. Střední vrstva pak tyto příkazy jen překládá na příkazy pro simulátory a přeposílá je. Pro podporu ladění programů v hardwaru se pomocí protokolu Nexus také zasílají zprávy laděnému procesoru. Zde se nabízí myšlenka, že by střední vrstva obsahovala objekt, který by měl definované rozhraní a sloužil by k překladu příchozích příkazů z prezentační vrstvy na data, které se zašlou buď simulátoru nebo

přes AUX port přímo do hardware. Takže by existovaly dvě implementace takovéhoto rozhraní, jedno by vracelo příkazy pro simulátory ve formě stávajícího XML protokolu a druhá implementace by vracela příkazy v binární formě odpovídající formátu zpráv definovaném ve standardu Nexus. Ovšem jak již bylo popsáno v páté kapitole, komunikace pomocí zpráv standardu Nexus je značně rozdílná od komunikace pomocí protokolu, kterým komunikuje střední vrstva se simulátorem. Je to dáno zejména z toho důvodu, že ne vždy existuje ekvivalentní Nexus zpráva ke zprávě pro simulátor. Jako příklad uvedu zprávu pro nastavení breakpointu, kdy pro simulátor stačí jediná zpráva obsahující všechny potřebné parametry. Ovšem pro nastavení breakpointu pomocí Nexus protokolu je nutné odeslat hned několik zpráv, nestačí jen jedna jako v případě zaslání zprávy simulátoru. Nastavení breakpointů probíhá skrze nastavení hned tří registrů (Breakpoint/Watchpoint control, Breakpoint/Watchpoint address a Breakpoint/Watchpoint data). Navíc nelze čekat ani na synchronní odpověď jakou zasílá simulátor, zprávy přes AUX port přicházejí asynchronně a jsou promíchány s dalšími asynchronními zprávami. Navíc v simulátorech není limitován počet breakpointů, ale v reálném hardware je počet breakpointů limitován počtem registrů pro breakpointy, kterých může být dva až osm. Střední vrstva si tedy musí udržovat interní reprezentaci těchto breakpointů, aby vědělo, které registry je možné použit a hlavně do kterého registru zapsat data odpovídající příkazu o vložení nového breakpointu. Střední vrstva také musí zpracovat asynchronní zprávy, které přichází přes AUX port. Například pokud dojde k zastavení procesoru, zařízení zašle jen zprávu Debug Status Message. Poté se musí pomocí zpráv pro čtení obsahu registru zjistit podrobnosti o důvodu zastavení procesoru. Lze tak učinit pomocí přečtení obsahu Development Status registru. Až pomocí těchto dat lze sestavit asynchronní zprávu pro prezentační vrstvu, že například došlo k zastavení procesoru z důvodu aktivace breakpointu.

Z důvodů této odlišnosti v protokolech navrhuji použít ve střední vrstvě abstraktní rozhraní, které bude obsahovat metody odpovídající příkazům prezentační vrstvy (např. vlož breakpoint, proveď krok programu, atd.). Toto rozhraní by pak bylo implementováno ve dvou různých třídách tak, aby jedna třída implementovala přeposlání příkazu simulátoru a druhá třída implementovala celou komunikaci pomocí protokolu Nexus. Metody tohoto rozhraní by byly volány poté, co by byl příkaz z prezentační vrstvy zpracován. Zpracování zpráv z prezentační vrstvy musí být implementováno mimo toto rozhraní také z toho důvodu, aby bylo možné zpracovat příkazy, které nejsou určené simulátorům nebo hardware. Navíc třída implementující zpracování příkazů pomocí rozhraní Nexus musí také obsahovat mechanismus na vyhodnocení výrazů. Při použití simulace se výrazy vyhodnocují v samotných simulátorech v ladicí knihovně. Vyhodnocení výrazů pro rozhraní Nexus musí být opět použito několika zpráv, které podle konkrétního výrazu vyhodnotí obsah paměti či registrů.

Na obrázku 8.1 je zobrazen diagram tříd, který zachycuje zjednodušenou strukturu a vztahy tříd ve střední vrstvě tak, aby odpovídaly navrženým změnám pro podporu ladění programů v hardware pomocí protokolu Nexus.



Obrázek 8.1: zjednodušený diagram tříd pro úpravy střední vrstvy aby podporovala i ladění programů v hardware pomocí AUX portu standardu Nexus

9 Závěr

V rámci této práce jsem se seznámil se základními architekturami víceprocesorových systémů na čipu a hlavně také s projektem Lissom, v rámci kterého je vyvíjeno prostředí pro návrh takovýchto systémů a programů pro tyto systémy. V rámci tohoto projektu byl vyvinut i ladicí nástroj pro jednoprocesorové systémy, se kterým jsem se detailně seznámil. Také jsem detailně prostudoval standard Nexus, který definuje požadavky pro jednotné nezávislé ladicí rozhraní pro víceprocesorové systémy na čipu. Podle požadavků definovaných v tomto standardu jsem pak navrhl a implementoval rozšíření funkcionality ladicího nástroje. Jednalo se zejména o podporu datových breakpointů. Dalším navrženým a implementovaným rozšířením tohoto nástroje byla podpora pro ladění víceprocesorových systémů na čipu v simulaci, které tvořilo hlavní cíl této práce. V rozšířeném nástroji je tak možné efektivně ladit programy pro víceprocesorové systémy na čipu. Uživatel si může pomocí ladicího nástroje při simulaci zvolit zastavení jednoho nebo všech procesorů pomocí breakpointů a po zastavení procesorů může jednotlivé procesory krokovat a zkoumat tak stav programů ve všech procesorech. K tomuto zkoumání může použít výpis hodnot registrů ale také i přímý přístup do paměti a zdrojů procesoru pomocí výrazů.

V rámci implementace rozšíření ladicího nástroje byly upraveny všechny vrstvy třívrstvé architektury projektu Lissom. Kromě ladicí knihovny byla upravena i střední vrstva a generátor simulátoru a na vrstvě uživatelského rozhraní byla rozšířena a upravena funkčnost příkazové řádky. Cíl implementovat ladicí nástroj pro víceprocesorové systémy na čipu byl tedy splněn.

V sedmé kapitole byl ladicí nástroj testován, zejména byl testován vliv ladicí knihovny na rychlost simulace. Některé rozšíření simulace znatelně zpomalují, a proto byla implementována možnost vygenerovat simulátor bez této podpory. Jedná se zejména o podporu datových breakpointů,

Pro budoucí rozšíření tohoto ladicího nástroje se počítá s implementací rozšíření pro ladění víceprocesorových systémů na čipu i do zásuvného modulu pro platformu Eclipse, který umožní ještě pohodlnější a uživatelsky přívětivější ladění programů. Implementace tohoto rozšíření bude spočívat vzhledem třívrstvé architektuře jen v implementaci rozšíření uživatelského rozhraní a implementaci rozšíření protokolu mezi střední vrstvou a uživatelským rozhraním tak, aby reflektoval změny a rozšíření v příkazové řádce.

Dalším možným rozšířením ladicího nástroje je rozšíření, kdy bude možno ladicí nástroj použít i pro ladění programů nejen v simulátorech, ale i programů, které poběží v prototypech hardware. Návrh tohoto rozšíření byl prezentován v kapitole 8.

Literatura

[1] Hruška, T.: Instruction Set Architecture C, Interní materiál projektu Lissom, Brno, 2009

[2] Kolektiv autorů: IEEE-ISTO 5001™-2003, The Nexus 5001 Forum™ Standard, IEEE-ISTO, 2003, Dostupné na URL: < http://www.nexus5001.org/>

[3] Wayne, W; Ahmed Amine, J.; Grant, M.: Multiprocessor System-on-Chip (MPSoC) Technology, IEEE Transactions on computer-aided design of integrated circuits and systems, 2008

[4] Kolektiv autorů: ARM AMBA Open Specifications, [cit. 2011-04-20], Dostupné na URL: < http://www.arm.com/products/system-ip/amba/amba-open-specifications.php>

[5] Kolektiv autorů: CoreConnect Bus Architecture, aktualizováno 1999-09-01 [cit. 2011-04-20], Dostupné na URL: <https://www-01.ibm.com/chips/techlib/techlib.nsf/products/CoreConnect_Bus_Architecture>

[6] Bereković, M.; Srolberg Hans-Joachim; Pirsch, P.: Multicore System-On-Chip Architecture for MPEG-4 Streaming Video, IEEE Transactions on circuits and systems for video technology, 2002

[7] Kumar, S.; Jantsch, A.; Soininen, J.; Forsell, M.; Millberg, M.; Öberg, J.; Tiensyrjä, K.; Heman ,A.: A Network on Chip Architecture and Design Methodology, IEEE Computer Society Annual Symposium on VLSI, 2002

[8] Mishra, P.; Dutt, N.: Architecture Description Languages for Programmable Embedded Systems, In IEE Proceedings on Computers and Digital Techniques, 2005

[9] Leupers, R.; Marwedel, P.: Retargetable code generation based on structural processor descriptions, Design Automation for Embedded Systems, 1998

[10] Freericks, M.: The nML machine description formalism, Technical Report TR SM-IMP/DIST/08, TU, Berlin CS Dept., 1993

[11] Hadjiyiannis, G.; Hanono, S.; Devadas, S.: ISDL: An instruction set description language for retargetability, In Proceedings of Design Automation Conference (DAC), 1997

[12] Zivojnovic, V.; Pees, S.; Meyr, H.: LISA - machine description language and generic machine model for HW/SW co-design, In IEEE Workshop on VLSI Signal Processing, 1996

[13] Comer, D. E.: Internetworking with TCP/IP. Prentice Hall, 1995

[14] Pechanec, J.: How the SCP protocol works, aktualizováno 2007-07-09, [cit. 2011-05-19], Dostupné na URL: <http://blogs.oracle.com/janp/entry/how_the_scp_protocol_works>

[15] Kolář, D.: Návrh výstupního formátu pro assembler a linker, Interní materiál projektu Lissom, Brno, 2004, aktualizováno 25.5.2010

[16] Rorsenberg, J. B.: How Debuggers Work. New York, USA, ISBN 0-471-14966-7.

[17] Kolektiv autorů: GDB: The GNU Project Debugger, aktualizováno 2009-11-10 [cit. 2010-12-23], Dostupné na URL: < http://www.gnu.org/software/gdb/documentation/>

[18] Kolektiv autorů: 1149.1-2001 - IEEE Standard Test Access Port and Boundary Scan Architecture, IEEE, 2001, Dostupné na URL: <http://standards.ieee.org/findstds/standard/1149.1-2001.html>

[19] Kane, G.; Heinrich, J.: MIPS RISC architectures, Prentice Hall, 1992

Příloha A

K diplomové práci je přiloženo paměťové médium (DVD) obsahující elektronickou verzi technické zprávy a zdrojové kódy aplikace.